Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2016

Pedro Miguel Ribeiro Piçarra

Armazenamento integrado em múltiplas clouds

Pedro Miguel Ribeiro Piçarra

Armazenamento integrado em múltiplas clouds

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Artur José Carneiro Pereira, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

"By three methods we may learn wisdom: First, by reflection, which is noblest; Second, by imitation, which is easiest; and third by experience, which is the bitterest."

— Confucius

o júri / the jury

presidente / president	**Professor Doutor André Ventura da Cruz Marnoto Zúquete** Professor Auxiliar da Universidade de Aveiro
vogais / examiners committee	**Professor Doutor Pedro Lopes da Silva Mariano** Professor Auxiliar Convidado na Faculdade de Ciências da Universidade de Lisboa (arguente)
	Professor Doutor Artur José Carneiro Pereira Professor Auxiliar da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Agradeço ao meu orientador, Professor Artur Pereira, pela oportunidade prestada, pelo projeto e reconhecimento e pelos conhecimentos adquiridos e transmitidos.

Agradeço também à minha família, amigos e colegas, em especial aos mais chegados. Todo o apoio recebido ao longo desta etapa foi e será fundamental para o meu bom desempenho profissional e crescimento de vida.

Palavras chave

Armazenamento remoto, Armazenamento em cloud, Armazenamento distribuído, Armazenamento integrado

Resumo

Com o avanço da tecnologia, o armazenamento de dados tem escalado de uma forma exponencial. Com o rápido desenvolvimento de técnicas e hardware para processamento de dados e capacidade de armazenamento de dispositivos, há cada vez mais uma menor preocupação no tamanho dos ficheiros gerados para um dado propósito. Simultaneamente, com o desenvolvimento de sistemas cloud e com o aumento das velocidades de transferência de dados na Internet, possuir cópias de dados armazenados em sistemas remotos tornou-se popular e até disponibilizada gratuitamente.

Hoje em dia, adquirir o computador está muito mais acessível a todos e o número de ficheiros por cada utilizador está a aumentar devido ao incremento massivo de documentos em suporte digital. Esse número leva a que a ocupação dos sistemas de armazenamento remotos disponibilizados para utilizadores atinja o seu máximo. Uma forma de contornar este problema é utilizando múltiplas clouds gratuitas. Porém, o crescimento do armazenamento disponibilizado gratuito, mesmo utilizando múltiplas contas, não acompanha o crescimento do tamanho de alguns tipos de ficheiros, como por exemplo os multimédia.

O problema reside, desta forma, em armazenar ficheiros cujo tamanho singular exceda o tamanho de um dado serviço de armazenamento cloud e no processo imprático de possuir vários serviços deste tipo na medida em que se apresentam individualmente separados perante o utilizador.

Desta forma, esta dissertação pretende o desenvolvimento de uma aplicação que tenha um funcionamento idêntico ao dos vários serviços de armazenamento cloud, integrando vários serviços deste género, fundindo-os num único, de uma forma transparente para o utilizador.

Keywords Remote storage, Cloud storage, Distributed storage, Integrated storage

Abstract

Technological progress led to the exponencially increase of data storage. With the quick development of data processing hardware and techniques and devices' storage capacity, files that are generated for a certain purpose have a tendency to become bigger without any concern attached. Simultaneously, with the speedy development of cloud systems and with the Internet data transfer speeds step-up, having backup data storaged in remote systems became popular and even provided free of charge.

Today, getting a computer is much more accessible to everyone and the number of files per user is growing due to the massive increase of digital format documents. That number leads to the maximum quota of the provided remote storage systems to end-users being achieved. One way around this problem is using multiple free of charge cloud services. However, the free of charge provided storage doesn't increase at the same rate of some files, for example, multimedia ones.

The problem lies in files that exceed the size of a certain cloud storage service and in the impractical process of having several services of this kind in a way that they present themselves individually separated to the user.

This way, this dissertation aims to develop an application that allows the identical features of several cloud storage services integrating diverse services of this kind, merging them into a single one, in a transparent way to the user.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação/Enquadramento

A necessidade de armazenamento de dados está presente desde a existência do primeiro computador. Desde cedo que se reconhecem os problemas gerados por esta necessidade, nomeadamente o espaço limitado dos dispositivos, a limitação de expansão devido a limitações físicas, a existência de um único ponto de falha e a consequente necessidade de backup, o que representa mais espaço ocupado.

Com o crescimento da tecnologia relativa ao armazenamento de dados e também da Internet, alguns limites começaram a ser ultrapassados. Os espaços disponíveis de armazenamento dos dispositivos aumentaram e o tamanho dos dispositivos diminuiu, as velocidades de escrita e leitura de dados também amadureceram, resultando num melhoramento geral desta tecnologia. Neste ponto de evolução, a solução do problema consistia no investimento e aquisição de um ou mais dispositivos físicos para comportar todos os dados necessários.

Com a evolução do mercado informático, os computadores tornaram-se cada vez mais acessíveis. O número de pessoas com um ou mais computadores começou a multiplicar-se.

Neste momento, gerou-se um problema. O avanço da tecnologia de processamento revelou imediatamente uma tendência do crescimento da maioria dos ficheiros (especialmente multimédia), uma vez que já era possível o processamento de mais dados e, consequentemente, melhores resultados.

A acessibilidade de ter um computador também tinha sofrido um grande acréscimo, o que levou a um número ampliado de computadores no mundo. O resultado final traduziu-se num número de computadores maior a produzir ficheiros de tamanho também maior para uma tecnologia de armazenamento mais desenvolvida mas não o suficiente para acomodar estas mudanças.

A situação teve um desenlace com o desenvolvimento da Internet. Devido ao crescimento de dados e à necessidade de troca deles entre utilizadores, a Internet cresceu, não só em número de utilizadores como também em capacidade de transferências e funcionalidades.

Com o decorrer do tempo, as larguras de banda da Internet foram tão desenvolvidas que

surgiu o conceito de armazenamento na nuvem (em *cloud*). Este tipo de armazenamento consiste num tipo de armazenamento remoto através da Internet. Quer em serviços públicos ou privados, os dados permanecem num ou mais servidores distribuídos geograficamente e acessíveis em qualquer local através da Internet.

Esta nova forma de armazenamento permitiu resolver o excesso de equipamentos adquiridos e concebeu um novo tipo de serviço onde um cliente pode ter um espaço de armazenamento adicional personalizado às suas necessidades para redundância de dados e/ou expansão do seu sistema de armazenamento atual a um custo reduzido, por um determinado tempo.

Ao existir tal tipo de serviço, a economia tornou-se imediatamente um fator decisivo para o estímulo de utilização do mesmo.

Com o seu desenvolvimento resultante do gigante número de novos utilizadores na Internet, foi disponibilizado, gradualmente, este tipo de armazenamento de forma gratuita, o que levou à enorme popularidade deste tipo de serviço, assim como o seu número de utilizadores.

Naturalmente, com a disponibilidade e forte competição neste tipo de indústria, um utilizador pode ter várias contas de variados serviços deste género, geralmente para gerar mais espaço disponível, gratuitamente.

Com base nesta sucessão de acontecimentos, surgiram serviços de gestão de múltiplos serviços de armazenamento remoto. Estes serviços são, na sua grande maioria, aplicações web que se conectam aos serviços *cloud* do utilizador (com a respetiva autorização) e permitem a listagem e operações sobre os ficheiros das diversas *clouds*.

Porém, mesmo com estes serviços de gestão múltipla, existe sempre a noção de separação desse espaço extra. As múltiplas contas são apresentadas separadamente, vulgarmente, em pastas, ainda que se possa transferir ficheiros de um serviço para o outro. Isto pode parecer óbvio e inteiramente apropriado para um grande número de utilizadores, pois os serviços são, de facto, distintos. No entanto, o conceito de integração e fusão transparente de vários dispositivos de armazenamento não é novo e responde à necessidade de armazenamento de ficheiros grandes.

Neste caso, o problema reflete-se no espaço total disponibilizado por cada serviço de armazenamento em *cloud*. Na situação exemplo em que o cliente tem 5 serviços *cloud*, cada um com 2 GB de espaço de armazenamento. Este espaço totaliza em 10 GB, porém, como está dividido em partições de 2 GB, cada ficheiro armazenado não poderá ter mais de 2 GB no seu tamanho total.

Para além deste problema, como referido anteriormente, os serviços de múltipla gestão são, usualmente, aplicações web acessíveis apenas através de um *browser*. Isto representa o impedimento de sincronização local automática. O download dos ficheiros nas diferentes *clouds* tem de ser feito de forma manual ou utilizando os clientes disponibilizados pelo respetivo serviço, para cada um desses serviços.

1.2 Objetivos

Esta dissertação propõe a integração de espaços de armazenamento de diferentes serviços de armazenamento em *cloud* num único espaço, de forma transparente. O utilizador deste espaço integrado não deve perceber diretamente que existem vários serviços onde os ficheiros são alojados, nem deve precisar de se preocupar com a forma como o alojamento é feito nesses serviços. Compete à solução a desenvolver distribuir os ficheiros pelos vários serviços da forma mais adequada.

Este paradigma de utilização de múltiplas *clouds* permite também resolver o problema de ficheiros demasiado grandes para o espaço disponibilizado por serviços gratuitos, tratando o espaço integrado como um único, sendo esse o limite para o tamanho dos ficheiros que lá se podem armazenar.

Pretende-se que a solução a desenvolver disponibilize o espaço integrado na forma de uma pasta local e que faça a sincronização automática, de forma bidirecional, com as *clouds* que utilize. Assim, por um lado, a solução deve comportar-se como as aplicações desktop normalmente disponibilizadas pelos serviços de armazenamento em *cloud*. Por outro lado, deve incluir as funcionalidades de gestão de múltiplas *clouds*, de forma transparente para o utilizador.

Embora, o uso de dois serviços de armazenamento em *cloud* sejam suficientes como prova de conceito, pretende-se a sua implementação usando três serviços.

Adicionalmente, tal como nos programas cliente dos serviços de armazenamento em *cloud* existentes, pretende-se dispor da funcionalidade de sincronização seletiva, entre outras opções básicas, como escolher a pasta onde sincronizar e associar ou desassociar contas. Para efeitos de seleção dessas funcionalidades, pretendia-se também uma interface gráfica para que o utilizador pudesse definir parâmetros dessas mesmas funcionalidades.

1.3 Concretização dos objetivos

No decorrer da dissertação, várias decisões foram tomadas de acordo não só com o percurso de desenvolvimento planeado como também com os obstáculos e erros encontrados na estrutura que foi definida inicialmente.

Um dos maiores obstáculos encontrados foi o tempo de concretização. Devido a este mesmo facto, só foi conseguida a implementação com 2 serviços de armazenamento *cloud*, ao contrário do que foi originalmente traçado. Também por este motivo, a interface gráfica e funcionalidades associadas à mesma ficaram por implementar. Tudo o resto definido como objetivo foi concretizado.

Outro aspeto fundamental a salientar foi a necessidade de adicionar novas características não definidas nos objetivos ao longo do tempo. No capítulo 5, refere-se um comportamento do programa dependente do número de *threads* de execução. Ao testarem-se vários valores para este parâmetro, sendo estes sempre justificados para o que se queria provar, obtiveram-se resultados que fizeram perceber a necessidade de adaptar o programa para que se possa ajustar este parâmetro de modo a obter um desempenho maior.

1.4 Estrutura da dissertação

Além deste capítulo, com a introdução, esta dissertação está organizada em mais 5 capítulos. No capítulo 2 faz-se uma descrição de trabalho já existente relacionado com o tema e objetivos desta dissertação, à data da mesma. São analisados alguns sistemas de armazenamento em *cloud* existentes, com base em estudos anteriores e em experiência prática própria. São ainda analisadas soluções já existentes que se aproximam funcionalmente da que é pretendida. Por último, analisa-se alguma tecnologia necessária a ter em conta neste tipo de aplicações, como por exemplo o método de autorização de acesso a serviços terceiros.

No capítulo 3, com base nos objetivos definidos, descreve-se a arquitetura geral da solução concebida, realçando-se os aspetos a ter em consideração na implementação de uma solução. De seguida, no capítulo 4, apresenta-se a sua solução implementada. Todas as funcionalidades estão descritas. Alguns mecanismos mais importantes e peculiares estão um pouco mais detalhados para que se perceba inteiramente o núcleo de funcionamento do programa.

No capítulo 5, a aplicação resultante é analisada, nomeadamente em termos das funcionalidades previstas e do seu desempenho relativamente ao funcionamento normal das aplicações cliente existentes. Surgem algumas conclusões a partir desta análise, com objetivo de responder às questões levantadas pelo comportamento da aplicação em determinadas circunstâncias (vários ficheiros, fragmentação, etc) e de perceber o que pode ser melhorado. É ainda analisada e testada uma possível alteração na implementação do programa que lhe fornece um melhor desempenho. Esta alteração surgiu em crítica destes resultados.

Finalmente, no capítulo 6, é efetuada uma análise crítica, tendo em conta, para além da aplicação resultante, as conclusões obtidas a partir dos resultados. Existe uma reflexão para o que funciona corretamente, o que é adequado para o funcionamento diário e regular para o utilizador comum e o que precisa de ser melhorado. É ainda referido possíveis direções para trabalho futuro, no melhoramentos do trabalho implementado, na concretização de objetivos que ficaram por fazer e na incorporação de novas funcionalidades que surgiram com o desenrolar deste trabalho.

Capítulo 2

Armazenamento na *cloud*

O armazenamento em *cloud* consiste num tipo de armazenamento em que os ficheiros se localizam em servidores remotos (geralmente de terceiros) que são acessíveis através da Internet pública. Estes servidores estão normalmente distribuídos geograficamente e organizados com um sistema de ficheiros eventualmente distinto do do utilizador, embora transparente para o mesmo.

Este tipo de armazenamento é uma tecnologia tipicamente prestada pelos denominados serviços de armazenamento *cloud*. Como tal, é também possível, para um dado utilizador, usar vários serviços deste género.

O acesso a esses ficheiros e respetivas operações são efetuadas através de aplicações desenvolvidas para o efeito. Geralmente, duas interfaces disponibilizadas por parte destes serviços para acesso e operações sobre os dados são aplicações web e *desktop*.

Este capítulo cobre vários tópicos relacionados com o armazenamento na *cloud*, desde prestadores de serviço às tecnologias envolvidas. Foi realizado um pequeno estudo sobre os serviços de armazenamento *cloud* existentes atualmente e uma análise sobre soluções para gestão de múltiplos serviços. Uma vez que, de acordo com os objetivos descritos em 1, se pretende de alguma forma aceder a dados de um determinado serviço de armazenamento em *cloud*, são ainda estudados alguns mecanismos de autorização de acesso a terceiros.

2.1 Serviços de armazenamento na *cloud*

Como referido acima, um serviço de armazenamento *cloud* é um serviço que fornece, ao utilizador, espaço de armazenamento em *cloud* incluindo uma ou mais interfaces de acesso aos dados. Atualmente, existe um grande número de serviços deste tipo. Esse grande número traduz-se numa grande competitividade entre os prestadores desse serviço. Todavia, de acordo com a finalidade do uso do serviço, existem *clouds* preferenciais no que diz respeito às funcionalidades oferecidas e prestadas.

O Google Trends revela que as *clouds* mais utilizadas e mais conhecidas pela maioria dos utilizadores são a Dropbox, a Google Drive e a OneDrive, como mencionado em [1]. Esse resultado mantém-se desde os últimos anos [10]. Assim, optou-se por fazer um estudo

destes serviços. O estudo focou-se no mecanismo de funcionamento da aplicação (eventos chave e operações principais), na comunicação e transporte de dados, no armazenamento de dados e nas APIs disponibilizadas. No final da análise de cada serviço individual, é feito um comentário geral com uma pequena comparação entre *clouds* relativamente a aspetos não óbvios recolhidos nas análises individuais.

2.1.1 Dropbox

A Dropbox é um sistema de armazenamento remoto desenvolvido pela Dropbox, Inc.[1] Este sistema consiste num espaço de armazenamento online que funciona como um disco rígido onde podemos adicionar e remover ficheiros através das aplicações fornecidas ou pela API disponibilizada. Estes ficheiros ficam disponíveis remotamente desde que se tenha uma ligação à Internet.

A Dropbox disponibiliza uma aplicação para *desktop* e uma interface web que permite a troca de ficheiros entre o disco local e o servidor em ambos os sentidos.[2] A aplicação *desktop* permite, adicionalmente, a sincronização automática, também bidirecional, desses ficheiros, quer total, quer parcial (sincronização seletiva).

Com base no que é oferecido nos dois tipos de interface e tendo em conta os objetivos desta dissertação, concluiu-se que para a concretização desses mesmos objetivos a interface web não é adequada. Por esse mesmo motivo, foi estudada a aplicação *desktop* para Linux.

Mecanismo

A instalação da aplicação cria uma pasta local onde são replicados todos ou apenas alguns dos ficheiros armazenados na *cloud*, consoante escolha do utilizador. Sempre que um ficheiro é criado, alterado ou removido nessa pasta ou no servidor remoto, a sua alteração é refletida na localização oposta.

As alterações de um ficheiro local são detetadas somente mediante alterações no corpo de dados desse ficheiro. Se um ficheiro existe, só será reportada e enviada uma alteração ao mesmo na localização oposta à origem da alteração se o seu corpo de dados for também alterado. O comando `touch`, por exemplo, não provoca qualquer sincronização, porque só altera a data de modificação e não o corpo de dados.

Existe ainda, para as alterações remotas, uma funcionalidade denominada *Lan Sync* que permite, quando existe uma alteração no servidor remoto, a verificação da existência do ficheiro em questão na rede local onde o utilizador está inserido.[4] Isto permite a obtenção do ficheiro localmente em vez dos servidores da Dropbox na Internet, para uma sincronização mais rápida e eficiente. A sua utilização faz sentido em ambientes com vários computadores ligados na mesma rede cujas contas Dropbox associadas tenham conteúdo partilhado entre si.

Por exemplo, numa empresa onde vários utilizadores têm cada um a sua conta Dropbox pessoal, pode estar inserida uma pasta partilhada com todos os utilizadores para conteúdos

[1] https://www.dropbox.com/

relacionados com a empresa. Um funcionário pode alterar um dos ficheiros, sendo submetido para o servidor remoto. Os clientes dos outros utilizadores recebem a notificação de alteração de ficheiro e, antes de ocorrer a sincronização através dos servidores da Dropbox na Internet, verificam a existência dele localmente. Ao encontrarem o ficheiro no computador do funcionário que fez a alteração, evitam a ligação à Internet e descarregam a partir daí. Essa funcionalidade pode ser ativada/desativada consoante a preferência do utilizador.

Entrando em aspetos mais técnicos, a aplicação *desktop* consiste num *daemon* que corre em segundo plano e é instalada na pasta *home*, pelo que não precisa de privilégios *root* para correr. Este *daemon* é o responsável pela sincronização de ficheiros (deteção de alterações, envio e receção de ficheiros). O pacote de instalação inclui uma extensão para o navegador de ficheiros Nautilus, não instalando o mesmo. Essa mesma extensão permite uma nova entrada no menu de contexto.

É ainda instalado um serviço em /usr/bin (Dropbox CLI) para gerir o *daemon* que fica na pasta home, quer a sua instalação, quer operações sobre ele. Isto faz com que a Dropbox esteja disponível para todos os utilizadores, cada um com um *daemon* individual. Esse serviço é o que permite a execução e paragem do *daemon*, incluindo a sua execução no arranque. Este serviço permite ainda a geração de links de um ficheiro específico bem como a exclusão de ficheiros do processo de sincronização.

O *daemon* é descarregado assim que se seleciona a opção -i (instalar) da Dropbox CLI. A análise do código do *daemon* não está disponível, pois o *daemon* já é descarregado como binário. A CLI está escrita em Python e é colocada em /usr/bin como um executável. Segundo alguma pesquisa online, o *daemon* monitoriza os ficheiros na pasta com a ajuda do próprio SO (por eventos) e com bibliotecas do Python.

O *daemon* da Dropbox está também programada em Python e usa um interpretador modificado com ofuscação de código. O código é ainda cifrado. A engenharia reversa é, portanto, difícil mas não impossível, pois já foi desenvolvido um programa que o faz.[8]

Este *daemon* parece usar o módulo de kernel do Linux *inotify* para deteção de criação, alteração ou remoção de ficheiros. O *inotify* notifica, aos que estão subscritos, eventos de ficheiros ou pastas de acordo com a sua subscrição. O *inotify* não suporta sub-árvores. Isto faz com que métodos automáticos e recursivos tenham de ser tratados a alto nível.

Comunicação e transporte

A aplicação da Dropbox está em constante comunicação com os servidores do serviço. Existe a troca de dados de controlo com o servidor de notificações periodicamente (108.x.x.x). Essa troca de dados de controlo também é despoletada por uma alteração na pasta.[6] O servidor para onde envia as alterações de ficheiros é diferente (45.x.x.x). [6] Todas as ligações a estes servidores são seguras com exceção do servidor que notifica alterações.

No que diz respeito ao envio de dados de ficheiros, só são enviadas alterações a um dado ficheiro se esse servidor não contiver uma versão desse exatamente igual, pois é criado um histórico e verificado se no servidor existe ou não esse ficheiro em questão [2][17].

Para além dos diferentes momentos em que a aplicação envia dados, é importante referir

que dados são efetivamente enviados. A Dropbox implementa *delta encoding*, isto é, só faz *upload* das partes modificadas do ficheiro, prevenindo que se tenha de fazer *upload* do ficheiro inteiro quando ocorre uma alteração [5].

As ligações TCP são feitas com *chunks* de 4 MB, pelo que não há um tamanho máximo de ficheiro na transferência, excluindo eventuais limitações do sistema de ficheiros na parte remota[5]. Quando existe o envio de vários ficheiros, existe ainda um conceito vantajoso a ter em conta - o *bundling*. O mecanismo de *bundling* consiste, quando existem vários ficheiros a serem enviados, num pipeline que torne o envio desse grupo de ficheiros o mais eficiente possível, evitando *overhead*. Em múltiplos ficheiros de pequena dimensão, o *bundling* é crucial pois para a inicialização de cada ligação TCP, existe um *handshake* inicial em que são trocados vários pacotes, o que exige algum tempo de espera até à transferência propriamente dita. Esta espera, derivada da ocorrência de acontecimentos descrita na frase anterior denomina-se por *slow start*. Ao evitar-se a criação de várias ligações TCP (uma ou mais por ficheiro) reutilizando uma já existente, esta-se a acelerar o processo potencialmente (reduz-se a troca de pacotes por cada ficheiro a enviar). A Dropbox implementa o *bundling* de ficheiros, reutilizando as conexões TCP que tem abertas [5].

É implementado ainda um modo de desduplicação (*deduplication*). A desduplicação é um mecanismo de compressão que evita dados duplicados. Quando é detetado que um ficheiro tem dados idênticos a outro(s) ficheiro(s), em vez dos dados serem duplicados, é colocada uma pequena referência para aquela zona de dados que nos indica que é idêntica à zona de dados de outro ficheiro, podendo, desta forma, ser mais eficiente tanto a nível de armazenamento como a nível de desempenho [5][2].

Finalmente, para a deteção de eventos remotos, o programa está sempre à escuta, usando um mecanismo de baixa latência. O servidor de eventos de alterações funciona com base numa ligação sempre ativa: o cliente da Dropbox manda um pedido para verificar se há alterações. Se sim, o servidor responde logo e o cliente faz outro pedido de imediato. Caso contrário, o servidor não responde até um máximo de 60 segundos. Aí é obrigado a enviar uma resposta, reportando que não há alterações, de modo a que a ligação não dê *timeout*. Caso haja uma alteração antes de atingir os 60 segundos, o servidor envia essa alteração. Isto repete-se ciclicamente [6].

Armazenamento

A Dropbox armazena os seus dados nos servidores da Amazon [6, 2]. Os ficheiros são armazenados também numa estrutura em árvore semelhante à presente no sistema operativo local. É ainda criado, para cada ficheiro, um identificador único para além do caminho.

API fornecida

A API da Dropbox, à data desta dissertação, está em constante desenvolvimento. São disponibilizadas as versões beta aquando do desenvolvimento de uma nova API. Entre

versões beta é importante referir que há funcionalidades que são retiradas de versão para versão, pelo que não é garantida retro-compatibilidade.

A API fornece métodos para autenticação e manipulação de ficheiros residentes na *cloud*, assim como para outros aspetos de gestão de conta, como por exemplo início de sessão. O protocolo de autorização usado no início da comunicação é o OAuth 2.0 [7].

Essa mesma API (versão 2 no momento da escrita desta dissertação) é disponibilizada em várias linguagens de programação.[2] Em Java, são usadas *streams* para operações sobre ficheiros quer em *upload* quer em *download* (este aspeto será relevante mais adiante). Para além das bibliotecas para várias linguagens de programação, é ainda disponibilizada uma API mais genérica com serviços web para o efeito das operações sobre ficheiros.

2.1.2 Google Drive

A Google Drive consiste num sistema de armazenamento *cloud* desenvolvido pela Google, Inc.[3] O sistema consiste num espaço de armazenamento online sobre o qual se pode fazer operações de leitura e escrita, tal como num disco físico através das aplicações disponibilizadas (tanto mobile como *desktop*) ou através da API fornecida, desde que se tenha uma ligação à Internet.

Semelhante à Dropbox, a Google Drive disponibiliza uma interface web e uma aplicação *desktop*. Identicamente, a interface web não permite a sincronização automática de ficheiros, pelo que foi estudada a aplicação *desktop* para o cumprimento dos objetivos definidos no capítulo 1. Atualmente, não existe um programa para *desktop* para Linux sendo que, para efeitos de compreender o seu funcionamento, foi estudada a aplicação para Windows.

Mecanismo

A aplicação do Google Drive funciona de um modo muito semelhante à da Dropbox. O funcionamento geral é igual, isto é, os ficheiros sincronizam-se bidirecionalmente quando há alterações, sejam locais ou remotas.

Toda o processo de instalação é muito semelhante ao da Dropbox. É criada numa pasta local destinada à replicação dos ficheiros e deteção de criação, alteração ou remoção dos mesmos. O processo fundamental de sincronização é também um *daemon* que corre em segundo plano. O *daemon* é ainda responsável por detetar alterações além de enviar/receber ficheiros.

Comunicação e transporte

No que toca a características de ligação entre a aplicação e os servidores, a comunicação é toda feita sobre HTTPS [5]. No envio de ficheiros, o processo de transporte é diferente do das outras *clouds*, na medida em que as conexões TCP acabam em nós periféricos da rede de servidores da Google, seguindo depois para os servidores de armazenamento pela rede

[2] https://www.dropbox.com/developers
[3] https://drive.google.com/drive/

privada da empresa [5]. Constata-se ainda a utilização de diferentes servidores dedicados para *download* e *upload*. No caso do *upload*, por exemplo, a Google Drive utiliza o endereço *upload.drive.google.com* [1].

Relativamente ao conteúdo enviado, a Google Drive possui, também como a Dropbox, um sistema que guarda histórico de ficheiros [17]. Porém, ao contrário da Dropbox, só é permitida a restauração de ficheiros, por parte do utilizador, ou seja, não é verificado no servidor, no momento de envio, se existe uma versão do ficheiro idêntica à que está a ser enviada.

O Google Drive usa *chunks* de 8 MB e não usa desduplicação como a Dropbox [5]. Não implementa *delta encoding*, ou seja, em vez de guardar apenas alterações ao ficheiro desde a última revisão, envia sempre o ficheiro todo. Também não implementa *bundling*, sendo que abre uma conexão TCP por ficheiro [5].

Tal como na Dropbox, os ficheiros são comprimidos antes de serem enviados. Isto torna o seu tamanho mais reduzido, pelo que diminui a latência e a carga de dados em ligações externas [5]. Contrariamente à Dropbox, antes dos ficheiros serem comprimidos para envio, existe um pré processamento dos mesmos para verificar a viabilidade da compressão. Por exemplo, se for colocada uma imagem JPEG falsa, cujo conteúdo não é informação sobre a dita imagem, a aplicação da Google Drive identifica e evita a sua compressão [5].

Armazenamento

Como referido na subsecção do Transporte, o armazenamento é feito em servidores da Google. O sistema de ficheiros da Google Drive é completamente diferente do que é usual observar-se. Não existe um conceito de caminho de diretórios direto.

Os ficheiros são armazenados como se não existissem pastas e as pastas são consideradas com um ficheiro. Cada ficheiro possui vários atributos, sendo um deles o *mime type*; no caso das pastas existe um *mime type* específico para as identificar. Cada ficheiro possui ainda um identificador único (que o identifica no sistema de ficheiros em que reside) e uma lista de pais (sendo estas as pastas em que se encontram).

API fornecida

A API fornecida pela Google Drive está, também, disponível para várias linguagens de programação e ainda em serviços web para um programa genérico. Para a corrente dissertação, foi considerada a biblioteca em Java[4]. Esta biblioteca disponibiliza métodos de autenticação, obtenção de metadados e manipulação de ficheiros, com distinção entre vários modos de acesso à *cloud* (modo de leitura, escrita, entre outros).

Na manipulação de ficheiros, a biblioteca não usa *streams* para *upload* e *download*. Em vez disso, usa um tipo de dados (presente na mesma) que abstrai o conceito de ficheiro, semelhante à classe `File` do Java, onde, internamente, cria *streams* para o efeito.

No início da comunicação e, mais uma vez, de um modo semelhante à Dropbox, o acesso ao servidor por parte do cliente é autorizado pelo protocolo OAuth 2.0 [9][11].

[4]https://developers.google.com/resources/api-libraries/documentation/drive/v2/java/latest/

2.1.3 OneDrive

A OneDrive, anteriormente conhecida como SkyDrive, é um sistema de armazenamento *cloud* desenvolvido pela Microsoft Corporation.[5] O sistema consiste num espaço de armazenamento online sobre o qual se podem realizar operações de leitura e escrita, através das aplicações fornecidas ou através da API disponível, sendo precisa uma ligação à Internet.

Semelhante aos outros serviços, a Microsoft dispõe de aplicações *desktop* para acesso aos ficheiros, assim como uma interface web. Mais uma vez, a interface web não permite a sincronização automática de ficheiros.

Uma vez que o programa é proprietário da Microsoft, não há versão para Linux. Foi estudada a aplicação para Windows tendo em conta os objetivos definidos no capítulo 1.

Mecanismo

Similarmente às outras *clouds*, a aplicação *desktop* cria uma pasta onde sincroniza os ficheiros local e remotamente, consoante haja alterações remota e localmente, respetivamente. A sincronização é, novamente, efetuada por um *daemon* em segundo plano que identifica também qualquer criação, alteração e remoção de ficheiros.

Comunicação e transporte

Todas as comunicações entre a aplicação e os servidores são feitas sobre HTTPS [5]. A OneDrive utiliza servidores para diferentes efeitos, separando-se em servidores de armazenamento e de controlo. Este serviço segue ainda uma política centralizada, similar à Dropbox, onde todo o tráfego chega aos servidores através da Internet pública, contrariamente à Google Drive que utiliza a sua rede privada para parte do percurso.

No envio, é executada a divisão do ficheiro em *chunks* para que, caso ocorra uma falha, o *upload* seja simplificado, enviando só as partes não concluídas anteriormente. A OneDrive, ao contrário dos outros serviços de armazenamento *cloud* mencionados anteriormente, reparte os ficheiros em *chunks* de tamanho variável.

Por fim, a OneDrive não implementa desduplicação, *bundling*, *delta encoding* ou compressão sobre ficheiros.

Um outro aspeto muito importante não evidenciado nas outras *clouds* é a limitação de velocidade. Enquanto que nas outras *clouds* estudadas nesta dissertação os limites são praticamente inexistentes, dependendo da velocidade de *upload* e *download* do utilizador, a OneDrive controla o *throughput* de *download* e limita essas velocidades para cerca de 10 Mb/s.

Armazenamento

A maioria dos servidores de armazenamento estão localizados dentro dos Estados Unidos da América. Existem vários servidores de controlo espalhados pelo mundo [5]. Este

[5] https://onedrive.live.com/

facto condiciona o tráfego, tanto em *upload* como em *download*, consoante a localização devido ao aumento da latência, uma vez que o acesso aos servidores é feito inteiramente pela Internet pública. Por exemplo, na Europa, um *upload* de um dado ficheiro demorará mais que nos Estados Unidos, assumindo uma ligação estável e com largura de banda compatível para o efeito.

API fornecida

A OneDrive fornece uma API em várias linguagens de programação - Python, C# e .NET. Admite ainda suporte para Android e iOS.[6] Tal como os outros serviços de armazenamento remoto, disponibiliza uma série de serviços web genéricos para que qualquer linguagem possa trabalhar sobre os ficheiros da Drive.

2.1.4 Análise comparativa

Depois de analisados três serviços de armazenamento *cloud*, faz-se aqui uma breve analise comparativa das suas características. Cada um apresenta vantagens nuns aspetos e desvantagens noutros.

A compressão inteligente do Google Drive, embora traga vantagem em termos de processamento (reconhece o que tem que ser comprimido), também traz desvantagem na medida em que, se tivermos muitos ficheiros de tamanho considerável, vai trazer um processamento maior e, consequentemente, um atraso no início do *upload* dos ficheiros.

Na Google Drive, a desvantagem de uma conexão TCP por ficheiro é compensada pelo transporte eficiente na sua rede privada. Assim como a sua compressão inteligente se cancela com a falta de *bundling*, o que torna o desempenho desta *cloud* boa mas não ótima [5].

A OneDrive, entre outros serviços não mencionados, sofrem por falta de *bundling*, compressão, *delta encoding* e/ou outras técnicas de otimização [5]. Em especial, a OneDrive limita as velocidades de *download*, o que torna o serviço ainda mais lento em sincronização.

A tabela 2.1 apresenta um resumo das técnicas de otimização presentes nos serviços analisado anteriormente que fundamentam as conclusões descritas acima. No geral, a Dropbox apresenta maior desempenho, pois implementa mais técnicas de otimização, o que se reflete em várias situações [5].

2.2 Gestão Multi-cloud

Existem vários serviços que gerem várias *clouds* simultaneamente. Estes serviços permitem a gestão de múltiplas contas *cloud* de serviços absolutamente distintos uma vez que, atualmente, os grandes serviços de armazenamento *cloud* disponibilizam APIs com serviços web que permitem leitura e escrita dos ficheiros presentes na *cloud* [21].

[6] https://dev.onedrive.com/readme.htm

Tabela 2.1: Comparação das técnicas de otimização dos diferentes serviços cloud

	Dropbox	Google Drive	One Drive
Repartição em chunks	4 MB	8 MB	Variável
Bundling	Sim	Não	Não
Compressão	Sempre	Inteligente	Não
Deduplication	Sim	Não	Não
Delta encoding	Sim	Não	Não

Desta forma, é possível, numa só aplicação, listar os ficheiros de várias *clouds* assim como fazer *download* ou *upload* deles. Consegue-se, além disso, movê-los de um serviço para o outro, sem a necessidade da intervenção de armazenamento local como meio intermediário, no caso das interfaces serem interfaces web.

No entanto, não há uma verdadeira integração. Todas as aplicações encontradas separam em pastas diferentes os diferentes serviços. Ou seja, podemos controlar todas as *clouds* no mesmo programa, mas o espaço de armazenamento é sempre individualizado. Não é oferecida ao utilizador uma abstração da alocação de ficheiros num só sítio porque essa gestão não é suportada.

Segue-se a análise dos serviços de gestão de múltiplas *clouds* que mais se aproximam do que é pretendido nos objetivos desta dissertação: a Cloudfuze e a OwnCloud. Essa análise tem em conta as capacidades dos serviços de gestão no que toca à apresentação de ficheiros e operações sobre os mesmos.

2.2.1 Cloudfuze

A CloudFuze, Inc é uma empresa fundada em 2012, com o objetivo de revolucionar a interligação entre armazenamentos *cloud*.[7] São oferecidos três produtos, uma versão para empresas e duas versões pessoais. Nesta dissertação, foi estudada a versão pessoal mais completa do programa.

O programa é pago mensal ou anualmente conforme a preferência do utilizador. O utilizador tem direito a 30 dias de teste.

Não é obrigatoriamente necessária nenhuma instalação local, pois é disponibilizada uma interface web. Todavia, também é disponibilizado uma aplicação *desktop* que permite com que as *clouds* sejam sincronizadas localmente, com um só programa.

O programa é acedido de duas formas possíveis: através do browser (interface web) ou através da aplicação *desktop*, que faz com que uma pasta local seja sincronizada. Dentro dessa pasta local os serviços são separados, cada um em sua pasta (ver figura 2.1).

[7] https://www.cloudfuze.com/

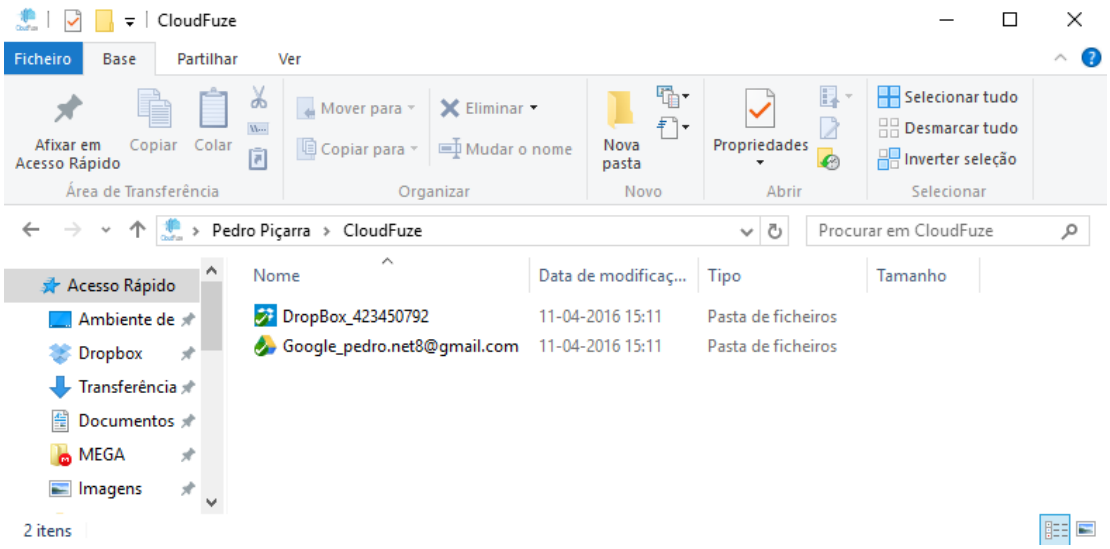


Figura 2.1: CloudFuze - Pasta local sincronizada

A aplicação *desktop* possui também um ícone na barra de tarefas para mostrar o estado do sincronizador e para opções relativas à execução fundamental do programa (*login*, *logout*, sincronizar e abrir pasta).

Note-se que, pelo facto das *clouds* serem separadas por pastas, não há forma de enviar um ficheiro cujo tamanho seja superior à *cloud* que possui mais espaço. A aplicação limita-se a gerir vários serviços de armazenamentos com políticas individuais, numa só aplicação.

Podemos adicionar múltiplas *clouds* através das duas interfaces, assim como operar sobre os ficheiros. Para adicionar as *clouds*, a aplicação fornece-nos uma lista de *clouds* suportadas (figura 2.2). O utilizador seleciona a que quer adicionar e é redirecionado para uma página onde tem de autorizar o acesso aos seus dados da *cloud* que pretende adicionar. Após este processo, a conta *cloud* fica associada ao CloudFuze.

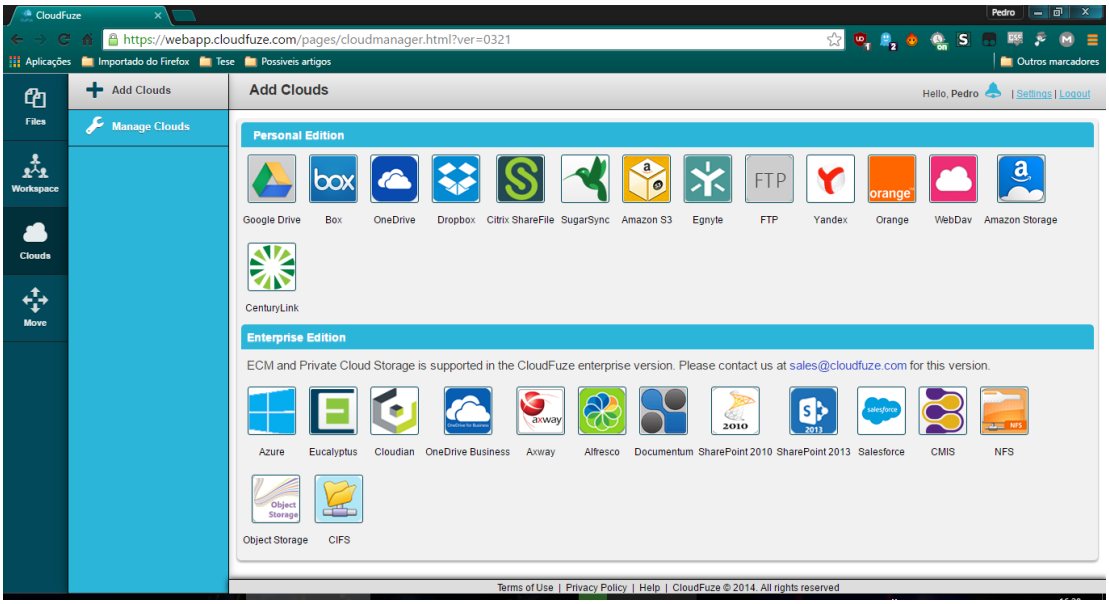


Figura 2.2: CloudFuze - Lista de *clouds* para adicionar

Tanto na pasta local como na aplicação web, as operações são intuitivas, uma vez que na interface web podemos escolher listar os ficheiros detalhadamente (figura 2.3) ou em ícones (2.4), tal como no explorador de ficheiros local. Assim, operações como arrastar para mover ficheiros são possíveis em ambas as interfaces.

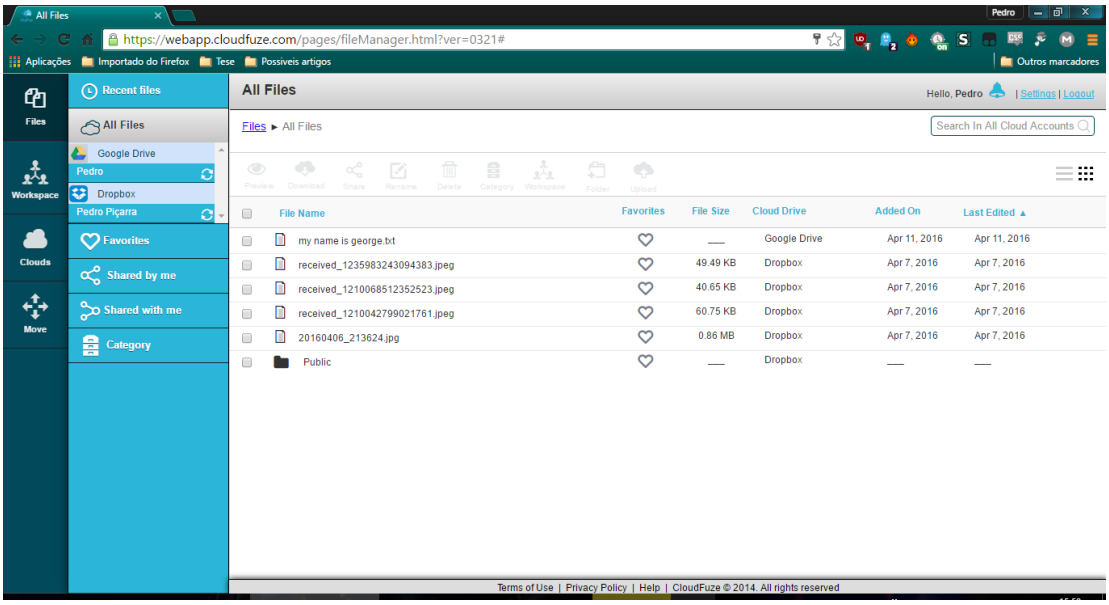


Figura 2.3: CloudFuze - Lista detalhada de ficheiros

Contudo, ao colocarmos os ficheiros dispostos por ícones, perdemos a informação a que

cloud pertencem, pois não há nenhum ícone ou outro elemento que represente isso, salvo na lista detalhada. Pode verificar-se esta situação nas figuras 2.3 e 2.4.

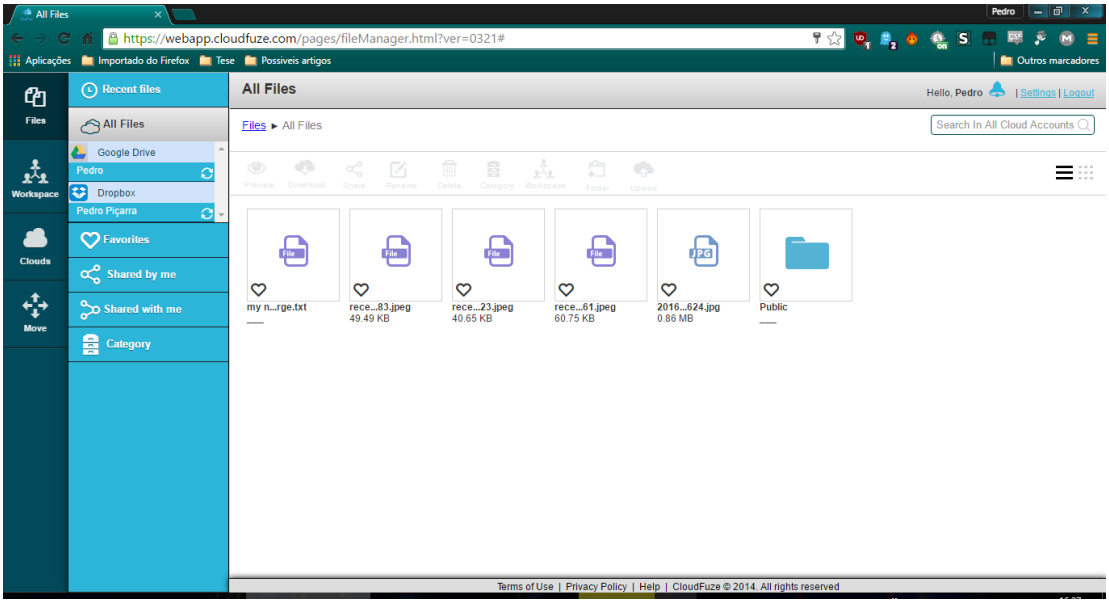


Figura 2.4: CloudFuze - Lista de ficheiros em ícones

2.2.2 OwnCloud

A OwnCloud é um serviço de armazenamento remoto alternativo aos serviços existentes, como por exemplo a Dropbox, Google Drive e OneDrive, desenvolvido por Frank Karlitschek, lançado em 2010 (interface web, a aplicação *desktop* foi lançada em 2012)[8]. A sua última versão, à data desta dissertação, foi lançada em Março de 2016. Nos últimos 6 anos foram lançadas 9 versões, pelo que o desenvolvimento é bastante ativo. Algumas das funções semelhantes às outras *clouds* foram desenvolvidas apenas no último ano. Este serviço não deve ser confundido com o OwnCloud empresarial, do mesmo autor.

Consiste numa aplicação semelhante à dos serviços referidos anteriormente e tem como particularidade o facto de ser open-source[9], gratuito e de se poder escolher onde alojar os ficheiros. Isto é, de se apontar à aplicação um servidor como local de armazenamento em vez de ter servidores próprios como anfitriões de armazenamento remoto [15]. Quer isto dizer que, na realidade, a OwnCloud possui uma aplicação cliente e uma aplicação servidor.

Qualquer pessoa pode ter um servidor próprio de armazenamento e associá-lo à aplicação *desktop* cliente. Basta, para isso, que o servidor de armazenamento corra a aplicação servidor e que se forneça o endereço web de acesso à aplicação cliente, seguindo as instruções que lhe são dadas.

[8]https://owncloud.org/
[9]http://github.com/owncloud

Este serviço indica também serviços de alojamento de dados terceiros que já correm uma variante da aplicação servidor do OwnCloud, tanto gratuitos como pagos, para os utilizadores que não possuam servidores de armazenamento próprios. Evidentemente, os ficheiros são também acessíveis via interface web, fornecida pela aplicação servidor.

Comparativamente aos serviços semelhantes analisados, as transferências são efetuadas em HTTP/WebDav. Não há transmissão das partes modificadas (*delta encoding*) e os *chunks* são de 10 MB, pelo que o tamanho máximo de um ficheiro a transferir é ilimitado, excluindo eventuais limitações ao sistema de ficheiros [16].

Apesar de ser open-source e apresentar algumas características possivelmente apetecíveis por muitos dos utilizadores, este serviço não é muito conhecido comparativamente aos serviços analisados anteriormente [10].

Integração com OwnDrive

No contexto desta dissertação, é relevante o facto de se combinar a OwnCloud com um servidor de armazenamento terceiro denominado de OwnDrive. Este permite-nos, para além do armazenamento remoto fornecido, adicionar como extensão de armazenamento contas de diferentes serviços *cloud* (como a Dropbox e Google Drive), servidores FTP, entre outros, de acordo com os protocolos implementados pela mesma.

No caso de adicionarmos um serviço *cloud* existente como extensão, por exemplo a Google Drive, precisamos de, nesse serviço *cloud*, criar uma aplicação na consola de desenvolvimento (acessível através das configurações do serviço cloud). Ao criar-se a aplicação, definir-se a política de acesso para todos os ficheiros e ativar-se as APIs necessárias, obtêm-se duas chaves para autenticação (este conceito é explicado em mais detalhe em 2.3.1). Posteriormente, na interface web da OwnDrive, adiciona-se a extensão de armazenamento da Google Drive, fornecendo as chaves e indicando o protocolo de autorização (neste caso, o OAuth 2.0). É então autorizado o acesso à Google Drive por parte da OwnDrive. Consequentemente, a aplicação da OwnCloud (que tem acesso à OwnDrive) também tem acesso a esse conteúdo e, a partir daí, os ficheiros e pastas da Google Drive pode ser sincronizado nos dois sentidos.

No entanto, é de notar que os ficheiros de cada armazenamento ficam separados, ou seja, o servidor de armazenamento (neste caso OwnDrive) é a pasta raiz. A extensão de armazenamento integrada (neste caso a Google Drive) fica numa pasta dentro da pasta raiz. Assim sendo, tem-se a mesma limitação da Cloudfuze - a transparência é limitada.

Existem outras limitações: para além do processo de integração de um serviço *cloud* externo ser mais complexo do que no Cloudfuze, ao adicionar-se a Dropbox em particular só é disponibilizado o protocolo de autorização OAuth (1ª versão), que se encontra obsoleta.

2.3 Mecanismos de autorização para serviços terceiros

Com a evolução dos diversos serviços remotos, como por exemplo armazenamento de ficheiros em *cloud*, e-mail, redes sociais, entre outros, foi também evoluindo a necessidade de interligação desses serviços. Por exemplo, é mais adequado armazenar-se todas as imagens recebidas via e-mail diretamente, sem intervenção do utilizador, no armazenamento *cloud*, do que fazer o *download* das imagens para o armazenamento local e, seguidamente, o seu *upload* para o serviço *cloud*. Isso significa necessariamente permitir o acesso de terceiros (neste caso o nosso e-mail) ao nosso serviço (neste caso o armazenamento em *cloud*).

Levanta-se o problema da autorização de acesso a esses mesmos serviços. O problema reside em dois aspetos: o programa que queremos interligar a um dado serviço teria de ter acesso a informação sensível (credenciais de acesso) e, com isso, poderia executar operações não desejadas; o programa poderia sofrer um ataque que poderia resultar em fuga de informação. Essa mesma fuga de informação poderia revelar passwords e outra informação sensível. É de realçar as passwords porque não só dão acesso ao serviço em si (combinadas com o nome de utilizador) como também podem dar acesso a outros serviços, tendo em conta que, e como é do conhecimento geral, muitas pessoas usam a mesma password para serviços diferentes com objetivo de não esquecer essa mesma password.

É também importante referir a diferença entre autenticação e autorização. A autenticação é o processo de identificação de um utilizador com um identificador único e um elemento secreto (por exemplo uma palavra-passe). Um processo de autenticação garante, após o mesmo, que se trata de um utilizador específico. A autorização é o processo de controlo de ações sobre um serviço. Não indica um utilizador específico por si só.

2.3.1 OAuth 2.0

O OAuth 2.0 é um protocolo de autorização web [18][19][14]. Tendo em conta os problemas referidos em 2.3, este protocolo disponibiliza mecanismos para evitar o fornecimento das credenciais de acesso aos serviços de terceiros.

Após análise desta mesma questão, chegou-se à conclusão que o grande problema seria ter apenas um método desafio-resposta (autenticação com as credenciais) que dava acesso a todas as funcionalidades. O problema solucionar-se-ia se tivéssemos todos os serviços diferenciados. Cada um teria como requisito permissões diferentes, o que requeria também autorizações diferentes. Porém isto acabaria por ser custoso e exaustivo para o utilizador, pois este teria de ser autorizado a cada operação diferente. No entanto, os serviços poderiam, ainda assim, estar diferenciados em termos de autorização (apenas utilizado por terceiros) e possuírem, contudo, um método de autenticação que permitiria o acesso a todas as funcionalidades (credenciais de acesso).

Um exemplo mais concreto, presente em [12], seria quando se compra um carro luxuoso. Geralmente são fornecidas pelo menos duas chaves, uma chave para o condutor e uma chave

para um manobrista [10]. A chave para o condutor permite operar a ignição e abrir todos os compartimentos do carro. A chave para o manobrista só deixa abrir a porta do condutor e operar a ignição. Todos os outros compartimentos (porta-malas, porta documentos, etc.) necessitam da chave do condutor.

Passando do exemplo para a situação real, é pedida a autorização de acesso ao serviço que queremos aceder, por parte do serviço terceiro. Esse pedido de autorização pode ser efetuado num servidor independente cujo propósito é a autorização. O serviço que vai ser acedido verifica o pedido e as funcionalidades requisitadas, e autoriza o acesso para essa aplicação terceira. É fornecido um token a essa mesma aplicação [3][18][14].

Esse token pode ter validade ou não e vai ser utilizado como objeto de autorização concedida (apenas às funcionalidades requisitadas) nas próximas sessões até expirar, no servidor onde a aplicação terceira quer aceder. Basta a aplicação terceira apresentar esse token para se comprovar fidedigna e poder realizar as operações que lhe foram permitidas.

Desta forma, não se fornecem dados que permitam o controlo total do serviço nem se fornecem dados sensíveis que, no caso de uma falha de segurança, permitam uma fuga de informação que comprometa toda a conta do serviço.

[10]Responsável por estacionar automóveis.

Capítulo 3

Projeto de uma solução

Atualmente existem diversos serviços de armazenamento *cloud*. No crescimento desta indústria, estes serviços passaram a disponibilizar uma modalidade gratuita e limitada em espaço, o que atraiu muito mais clientes para este mercado.

Com a expansão destes serviços e a multiplicidade de contas de serviços distintos por parte do mesmo utilizador, surgiram serviços terceiros para gestão de múltiplos serviços *cloud*. Estes serviços consistem em aplicações web que se conectam aos serviços *cloud* do utilizador e, através da sua interface web, listam e permitem operações sobre os ficheiros das *clouds* associadas. Porém, estes serviços apresentam os vários serviços *cloud* separados em pastas diferentes, permitindo, quanto muito, a transferência de ficheiros de uma *cloud* para outra.

Embora estes serviços disponibilizem uma gestão conjunta de múltiplas *clouds*, não são uma solução integrada. Por um lado, compete ao utilizador distribuir os seus ficheiros pelas várias *clouds*. Por outro lado, o tamanho dos ficheiros está limitado pelo espaço disponível em cada *cloud* e não pelo espaço total. Supondo, por exemplo, que um cliente tem 5 serviços *cloud*, cada um com 2 GB de espaço de armazenamento, o espaço totaliza 10 GB, porém, como está dividido em partições de 2 GB, cada ficheiro armazenado não poderá ter mais de 2 GB no seu tamanho total. Além disso, a maioria destes serviços são web, pelo que não permitem a sincronização local automática, ou seja, o *download* dos ficheiros tem de ser feito de forma manual ou utilizando os clientes *desktop* de cada *cloud* individualmente.

Nesta dissertação o objetivo é a obtenção de uma solução integrada (ver figura 3.1). Pretende-se visualizar o espaço de armazenamento de diversas *clouds* como uma só partição de uma forma transparente para o utilizador e realizar a sincronização local automática de pastas. O objetivo é, desta forma, ter uma solução semelhante aos clientes *desktop* já existentes para as diversas *clouds* que tenha os ficheiros das várias *clouds* associadas a este programa numa única pasta local, sincronizados. Quando se coloca um ficheiro, deverá ser identificado se o ficheiro pode ser enviado na totalidade para uma *cloud* (isto é, se existe uma *cloud* com espaço disponível para esse ficheiro) ou se tem de ser dividido de forma a aproveitar o espaço de mais *clouds*. No caso de ser dividido, este facto tem de abstraído para o utilizador. Isto é, para o utilizador, o ficheiro está naquela pasta local

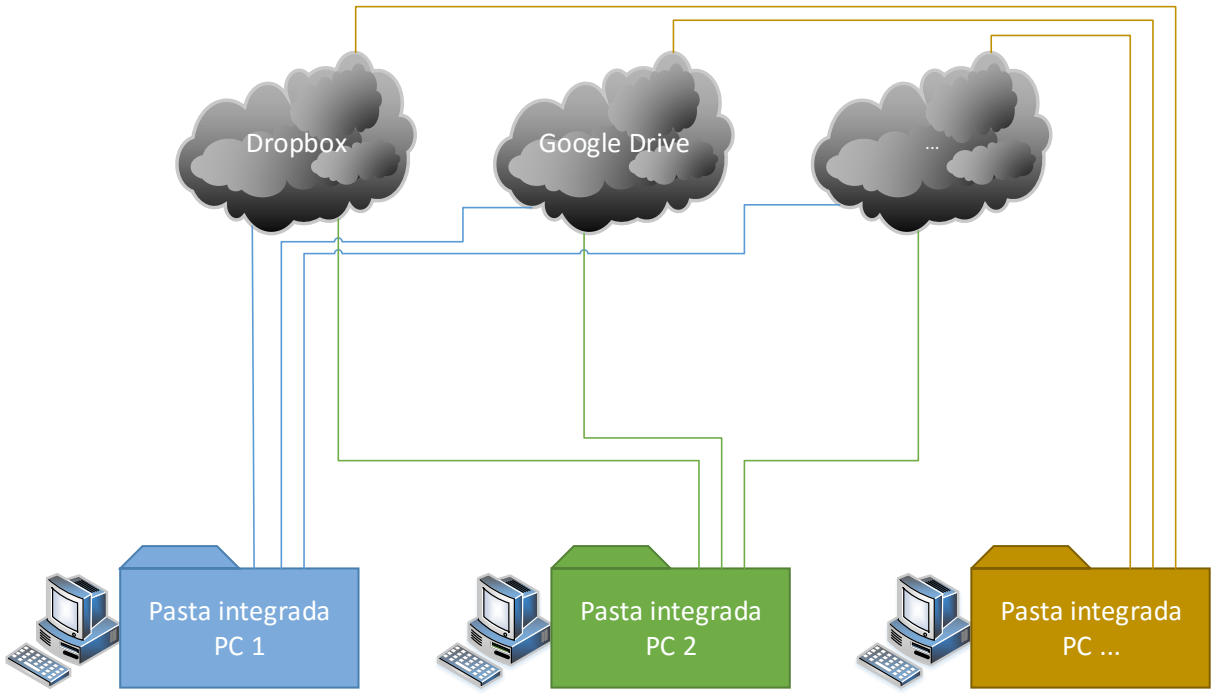


Figura 3.1: Visão integrada de várias *clouds*

e no serviço *cloud* conjunto remoto, não interessando propriamente em que *clouds* ou em quantas partes. O utilizador vê o ficheiro como um todo na sua pasta local, tal como se usasse apenas uma *cloud*.

Foi observado no capítulo 2, na análise dos diferentes serviços de armazenamento em *cloud*, que as interfaces web não permitem a sincronização automática dos ficheiros. Por esta razão e porque, como referido anteriormente, o que é pretendido é uma solução semelhante aos clientes *desktop* já existentes das diversas *clouds*, a solução basear-se-á numa aplicação *desktop*.

3.1 Arquitetura global

Para a realização de todas as funcionalidades referidas acima, temos de ter em conta dois fluxos de execução: *download* e *upload* de ficheiros. Em consequência da ocorrência de alterações a um ficheiro num dado computador, é necessário refletir essas alterações na multi-cloud, levando a operações de *upload*. Essas alterações, por sua vez, obrigam à alteração das cópias desses ficheiros noutros computadores, levando a operações de *download*. Ao mesmo tempo, é necessário garantir que as alterações na multi-cloud não se reflitam sobre o computador que provocou essas alterações.

Na figura 3.2 apresenta-se a arquitetura geral da solução. Foram identificadas algumas entidades cruciais na organização estrutural da aplicação de modo a cumprir os objetivos definidos anteriormente. Primeiramente, é necessário um módulo para gestão da aplicação.

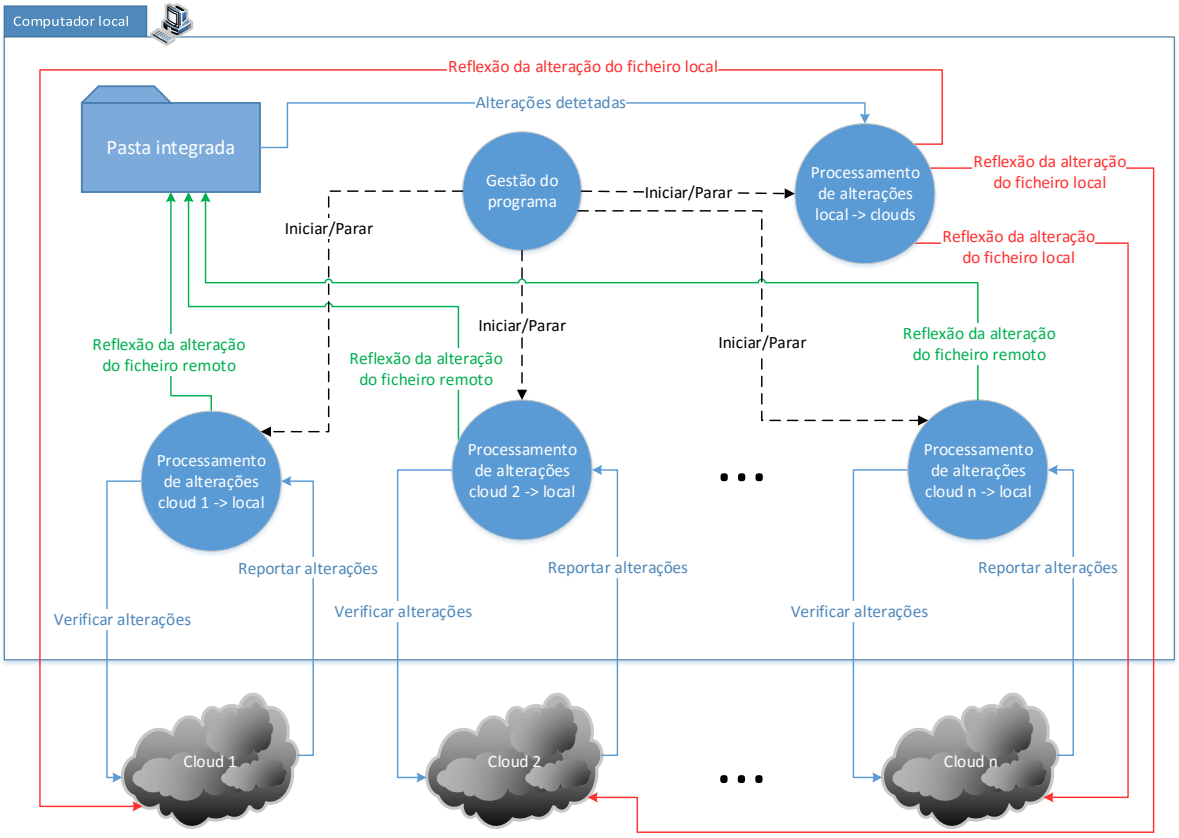


Figura 3.2: Arquitetura global da solução, para cada computador local

Este módulo será responsável por gerir os outros componentes restantes - iniciá-los e pará-los quer automaticamente quer por ordem do utilizador. Fundamentalmente, este módulo é o primeiro a correr no início da execução da aplicação e é o designado para iniciar os outros módulos.

De seguida, temos o módulo de processamento de alterações no sentido local → *clouds*. Este módulo irá ser responsável por agir conforme as alterações a ficheiros locais. É este mesmo componente que irá criar, modificar ou apagar os ficheiros em cada *cloud* e dividi-los previamente se necessário.

Por fim, temos a entidade de processamento de alterações no sentido *cloud* → local. Existirá, para cada *cloud*, uma entidade deste género para verificar alterações na respetiva *cloud* e fazer refletir essa alteração na pasta local. Ou seja, sempre que haja uma alteração numa dada *cloud*, a entidade dessa mesma *cloud* irá detetá-la e criar, modificar ou apagar o ficheiro na pasta local sincronizada.

3.2 Alterações locais

No que diz respeito a alterações de ficheiros locais, tem de haver um mecanismo de monitorização que permita detetar essas alterações. Este mecanismo permitirá o envio de ficheiros sempre que sejam criados ou modificados localmente e a sua remoção remota sempre que sejam apagados localmente.

Existem dois métodos possíveis para essa monitorização:

1. *Polling* - Verificar constantemente a pasta e sub-pastas por alterações. É altamente compatível mas tem, geralmente, má performance, uma vez que consome muitos recursos de processamento só para realizar essa operação ciclicamente.

2. Notificação por evento - A aplicação subscreve-se a um canal de comunicação e fica à escuta num processo de baixa latência, sempre que há uma alteração a um ficheiro, é notificado e pode atuar conforme a alteração. Este método tem bom desempenho, uma vez que a aplicação só necessita de recursos quando é notificada de alterações. É, porém, menos compatível pois para ser usado é necessário que o sistema operativo suporte este sistema de notificação, uma vez que é este que gere as notificações. No entanto, este último fator é menos significativo, pois a grande maioria dos sistemas operativos já possui esta característica.

O melhor método a utilizar será, então, o de notificação por evento. O Linux possui o módulo do kernel *inotify* [1]. Este módulo pode ser acedido por bibliotecas externas dependendo da linguagem de programação utilizada.

3.3 Alterações remotas

Para além da monitorização local, referida em 3.2, que tratará de fazer *upload* de ficheiros que forem criados, modificados localmente e apagar ficheiros remotos que foram apagados localmente, existe ainda a necessidade de ter outro mecanismo que trate do fluxo de *download*, isto é, a sincronização local de ficheiros alterados remotamente.

Sempre que ocorre uma alteração remota, a aplicação terá de ser notificado e transferir esse(s) ficheiro(s).

Enquanto que na monitorização local só é preciso um fluxo de execução, na monitorização remota precisamos de uma *thread* por cada serviço *cloud*. Isto deve-se à possibilidade de ocorrerem alterações em diferentes *clouds*, em simultâneo.

Tal como nas alterações locais, as alterações remotas, dependendo das funcionalidades disponibilizadas por um dado serviço *cloud*, podem ser monitorizadas em dois métodos - *polling* e notificação por evento. No caso do método de *polling*, não haveria problema em colocar uma única *thread* de execução, pois com o método de *polling*, há sempre uma resposta quando se verifica por alterações Após essa resposta é possível alternar a verificação de alterações entre *clouds*.

[1] http://man7.org/linux/man-pages/man7/inotify.7.html

Já no caso da notificação por evento, como o método de verificação é bloqueante (por esperar pelo evento) e nem sempre há resposta imediata, não faria sentido usar uma única *thread* para a verificação de alterações em todas as *clouds*, pois teria de haver alterações numa *cloud* para se proceder à verificação da seguinte.

Deste modo, decidiu-se usar uma abordagem multi-thread uma vez que esta abrange os dois métodos descritos acima. Assim, para cada *cloud*, existirá uma *thread* dedicada a detetar alterações e fazer refletir a alteração no lado local.

3.4 Circularidade

Um aspeto a ter em conta é que a concorrência entre os dois fluxos de alterações (local e remotos) pode provocar problemas de circularidade. Este facto deve-se a cada vez que há uma alteração local ou remota, precisarmos de sincronizar o ficheiro remoto ou local, respetivamente.

Quando há uma alteração a um ficheiro, é necessário sincronizar no lado oposto onde ocorreu a alteração. Ao sincronizarmos, vai haver uma notificação de que o ficheiro a sincronizar foi alterado. Isto vai desencadear o envio do ficheiro novamente onde ocorreu a alteração, quando na verdade não foi alterado, e tudo se repete.

Tomemos como exemplo o que aconteceria se um ficheiro fosse modificado remotamente:

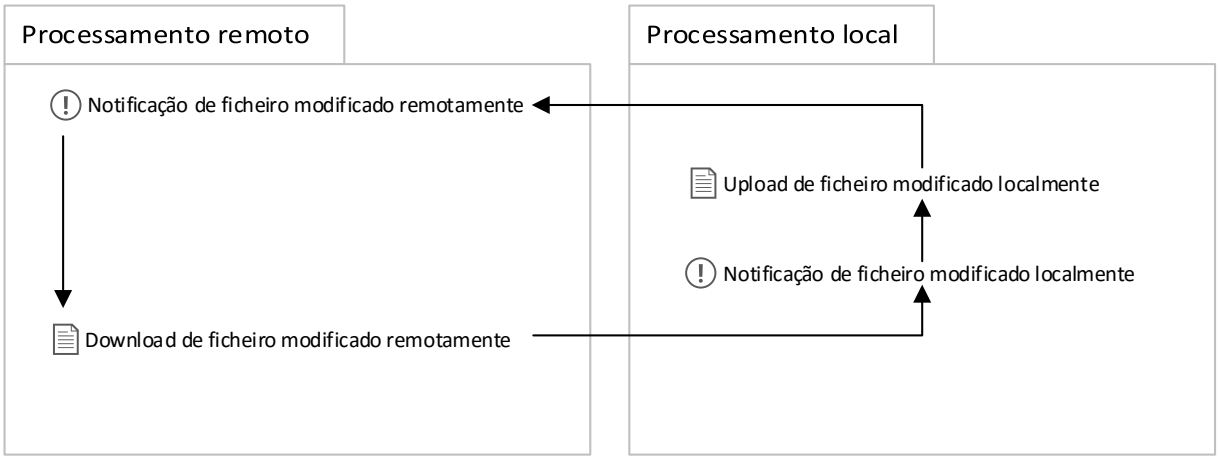


Figura 3.3: Sincronização cíclica a evitar

O fluxo de ações nessa situação (descritos na figura 3.3) seria, portanto

1. Notificação de ficheiro modificado remotamente
2. *Download* de ficheiro modificado remotamente
3. Notificação de ficheiro modificado localmente
4. *Upload* de ficheiro modificado localmente

5. Notificação de ficheiro modificado remotamente
6. *Download* de ficheiro modificado remotamente
7. ...

Este exemplo é também válido para o fluxo inverso em que um ficheiro é modificado localmente.

Para evitar esta situação, é necessário saber quando um ficheiro vai ser sincronizado e ignorar eventos gerados a partir dessas alterações.

Terá, então, de existir uma lista de ficheiros que se traduza no próximo evento gerado para esse ficheiro ser ignorado.

Quer isto dizer que as threads de monitorização local e remota terão de comunicar entre si (através de uma estrutura de dados partilhada) para reportarem que alterações de ficheiro deverão ser ignoradas, de forma a evitar ciclos e transmissões de dados desnecessárias.

Da mesma forma que há situações em que os eventos devem ser ignorados, há ficheiros de funcionamento interno da aplicação que nunca devem ser enviados.

Deve existir, desta forma, uma lista de ficheiros ignorados em permanência.

3.5 Serviços *cloud* e APIs

Para aceder aos serviços *cloud* é necessária uma interface que nos disponibilize métodos para podermos operar sobre os ficheiros residentes nesses serviços.

Para os serviços *cloud* a utilizar neste programa, é necessário ter em conta não só os serviços oferecidos (essencialmente o espaço de armazenamento oferecido) mas também a interface fornecida para operações sobre ficheiros remotos.

A aplicação deverá, ainda assim, ser modular e suportar qualquer tipo de *cloud* que respeite um conjunto de métodos essenciais ao funcionamento da aplicação, isto é, seguir uma API em que seja possível utilizar implementar o funcionamento para qualquer *cloud*. Essa API deve ter métodos que irão ser executados a cada alteração diferente de ficheiros (criação, modificação e remoção).

Assim, adicionar serviços *cloud* fica mais simples, uma vez que só temos de tratar dos métodos necessários para respeitar a API.

A API que cada cliente *cloud* terá de seguir constituir-se-á em métodos de:

- Insert - Inserir ficheiro na *cloud*
- Update - Atualizar ficheiro na *cloud*
- Delete - Apagar ficheiro na *cloud*
- Get - Obter ficheiro da *cloud*
- OnRemoteChange - Procedimento a realizar quando detetar alterações remotas (criar, modificar ou apagar o ficheiro remoto no disco local)

- Init - inicialização das conexões e autenticação

3.6 Base de dados e registo local

Na possibilidade de termos a aplicação a correr em vários dispositivos, temos de saber qual é a versão mais recente dos ficheiros e que ficheiros estão em falta para assegurar consistência de dados.

Neste caso, temos de ter uma base de dados que contenha a informação de quais os ficheiros existentes na pasta e um identificador para cada um que indique o quão recente é.

Como usamos mais que um serviço *cloud*, temos de indicar também em que serviço (ou serviços) está armazenado cada ficheiro.

Faz sentido, neste caso, usar uma base de dados e não carregar tudo em memória, pois, tanto quanto sabemos, podemos ter uma quantidade enorme de ficheiros e metadata dos mesmos. A aplicação irá ser concebido para correr em segundo plano, pelo que a sua marca deverá ser a menor possível para não afetar as outras aplicações.

A base de dados terá de ter dois níveis para uma simplicidade e modularidade em futura programação. Terá um primeiro nível geral, que lista todos os ficheiros e indica em que serviço (ou serviços) estão. Depois ramificar-se-á num segundo nível que se separa em várias *clouds*, isto é, cada *cloud* terá uma base de dados para os seus ficheiros devido ao formato diferente das informações que são guardadas, provenientes de cada serviço *cloud*, relativamente a cada ficheiro (nome, caminho remoto, revisões, etc.).

3.7 Divisão de ficheiros

Os serviços *cloud* podem possuir diferentes tamanhos de espaço de armazenamento. Num serviço normal de armazenamento em *cloud* o tamanho de um ficheiro estaria limitado a um dos seguintes:

1. O espaço disponível da *cloud*.

2. O tamanho máximo limitado pelo serviço (por questões de otimização de transferências de ficheiros ou limitações da arquitetura do serviço *cloud*, por exemplo sistema de ficheiros)

Nesta dissertação, queremos ultrapassar essa barreira. Para isso, utilizámos a mesma estratégia utilizada em transferência de ficheiros grandes - a divisão em partes. No caso de um ficheiro ultrapassar o tamanho máximo de um dos pontos referidos acima, o ficheiro terá de ser dividido em várias partes de modo a poder ser armazenado em múltiplas *clouds* pelo que terá de existir um método de divisão e junção de ficheiros. As partes dos ficheiros divididos irão ser geradas numa diretoria especial, com um nome reservado. Essa diretoria fará parte dos ficheiros ignorados a enviar.

Remotamente, para representar o ficheiro, cria-se uma pasta com o nome exato do ficheiro. Localmente o ficheiro deve ficar inteiro, as partes são enviadas para dentro da pasta criada remotamente.

Se um ficheiro for alterado remotamente, terá de se reconstruir o ficheiro localmente. Isto obriga a fazer *download* das partes todas, unir e depois apagar as partes, visto que já não são necessárias. Se o ficheiro local for alterado, divide-se novamente em partes e envia-se de novo para a localização remota.

É ainda essencial que haja um mecanismo que sinalize a aplicação de quais os ficheiros que são fragmentados para que, quando forem detetados novos ficheiros numa dada *cloud*, a aplicação perceba se esses ficheiros são ou não partes de um ficheiro fragmentado e, desta forma, evite transferi-los como um ficheiro normal mas sim como um ficheiro fragmentado, reconstruindo o mesmo.

3.8 Balanceamento

Como referido na secção 3.7, na situação de um *upload* de um ficheiro num sistema de armazenamento de ficheiros *cloud* comum, a aplicação só tem de ter em conta o espaço de armazenamento disponível e o tamanho máximo de transferência, se existir, para a *cloud* em questão. Se o tamanho do ficheiro não ultrapassar estes critérios, o *upload* pode ser realizado de forma normal.

No caso da solução desenvolvida no âmbito desta dissertação, para além de termos em conta estes dois aspetos, há a necessidade de uma análise acrescida devido ao facto de utilizarmos múltiplos serviços *cloud*.

Para além dos pontos referidos acima, o *upload* poderá, opcionalmente, ser influenciado pelo espaço ocupado em cada serviço *cloud* de forma a balancear a carga de ficheiros nas diferentes *clouds*.

Desta forma, se for decidido que um ficheiro é repartido em fragmentos, cada fragmento deverá ser colocado numa *cloud* diferente, rodando entre elas. Como os fragmentos têm o mesmo tamanho (com a exceção do último), assegura-se o balanceamento em termos de espaço ocupado.

Finalmente, quando um ficheiro é enviado inteiro, terá de ser analisado qual a *cloud* que contém mais espaço livre e, desta forma, fazer *upload* do ficheiro para a mesma.

3.9 Limitações

A solução apresentada neste capítulo apresenta algumas limitações.

Os ficheiros da pasta integrada estão distribuídos pelas várias *clouds*, pelo que o conteúdo de cada *cloud* apenas representa uma parte desse todo. A utilização das interfaces individuais (quer web, quer *desktop*) de acesso às *clouds* deve ser cautelosa. A criação de um novo ficheiro assim como a alteração de nome/caminho de um ficheiro já existente podem causar a colisão em nome/caminho com um outro ficheiro já existente na pasta integrada.

Para além disso, a remoção de fragmentos de ficheiros divididos representa um problema, pois a remoção de um fragmento pode causar situações de erro na aplicação e/ou corrupção de dados nos ficheiros reconstruídos. Finalmente, a alteração de ficheiros fragmentados é uma potencial causa de corrupção se essa alteração não for consistente.

Por fim, esta solução traz a necessidade de adotar nomes de ficheiros ou pastas únicos e exclusivos para o funcionamento da aplicação. Um exemplo, nesta arquitetura, é a diretoria temporária para processamento de ficheiro. Esta terá de estar presente na pasta partilhada e, consequentemente, mais nenhum ficheiro ou pasta poderão ter esse nome.

Capítulo 4

Implementação

Neste capítulo descreve-se o trabalho realizado na implementação de uma solução de acordo com a solução arquitetural definida no capítulo anterior. Considerando que a Dropbox e a Google Drive são os dois fornecedores de armazenamento *cloud* mais utilizados, optou-se por os usar na construção inicial dessa solução. Havia, no entanto, intenção de também usar o OneDrive. Tal não veio a acontecer por falta de tempo. Mesmo assim, o uso de dois fornecedores de armazenamento *cloud* permite mostrar a validade da solução apresentada.

A linguagem Java foi usada como suporte ao desenvolvimento. Dois fatores contribuíram decisivamente para essa escolha. Por um lado, o autor possuía bastante experiência de programação em Java. Por outro lado, os mecanismos necessários para a interação com as *clouds* escolhidas estavam disponíveis para Java.

4.1 Estrutura geral

Tal como mostra a figura 4.1 e de acordo com a solução arquitetural apresentada no capítulo anterior, a implementação assenta em $n + 1$ threads, sendo n o número de *clouds* utilizadas. Existe uma thread dedicada à deteção de alterações locais e à propagação dessas alterações para as *clouds*. Existe, por cada *cloud*, uma thread dedicada à deteção de alterações remotas e à propagação dessas alterações para a pasta integrada.

Para facilitar o trabalho de inserção de novas *clouds*, foram identificadas as operações básicas de interação com as *clouds* e foi definida uma API para a solução. Para cada *cloud*, os métodos dessa API são implementados usando os métodos da API da *cloud*.

Sendo uma solução multi-thread há que garantir a exclusão mútua no acesso a ficheiros, de modo a prevenir corrupção dos seus dados. Optou-se por usar a classe `ReentrantLock` do pacote `java.util.concurrent.lock`. A razão da escolha prende-se com a experiência do autor na sua utilização.

Para a deteção de alterações locais, o Java disponibiliza um pacote chamado NIO (Non-Blocking I/O). Este pacote disponibiliza métodos não bloqueantes para o tratamento dos ficheiros. Uma das funcionalidades deste conjunto de métodos é a deteção de eventos sobre

os ficheiros (criação, modificação e remoção dos mesmos). Estes métodos já suportam os métodos nativos do sistema operativo para a notificação desses eventos. Caso o Java não detete nenhum desses métodos nativos, recorre ao *polling* para monitorizar alterações.

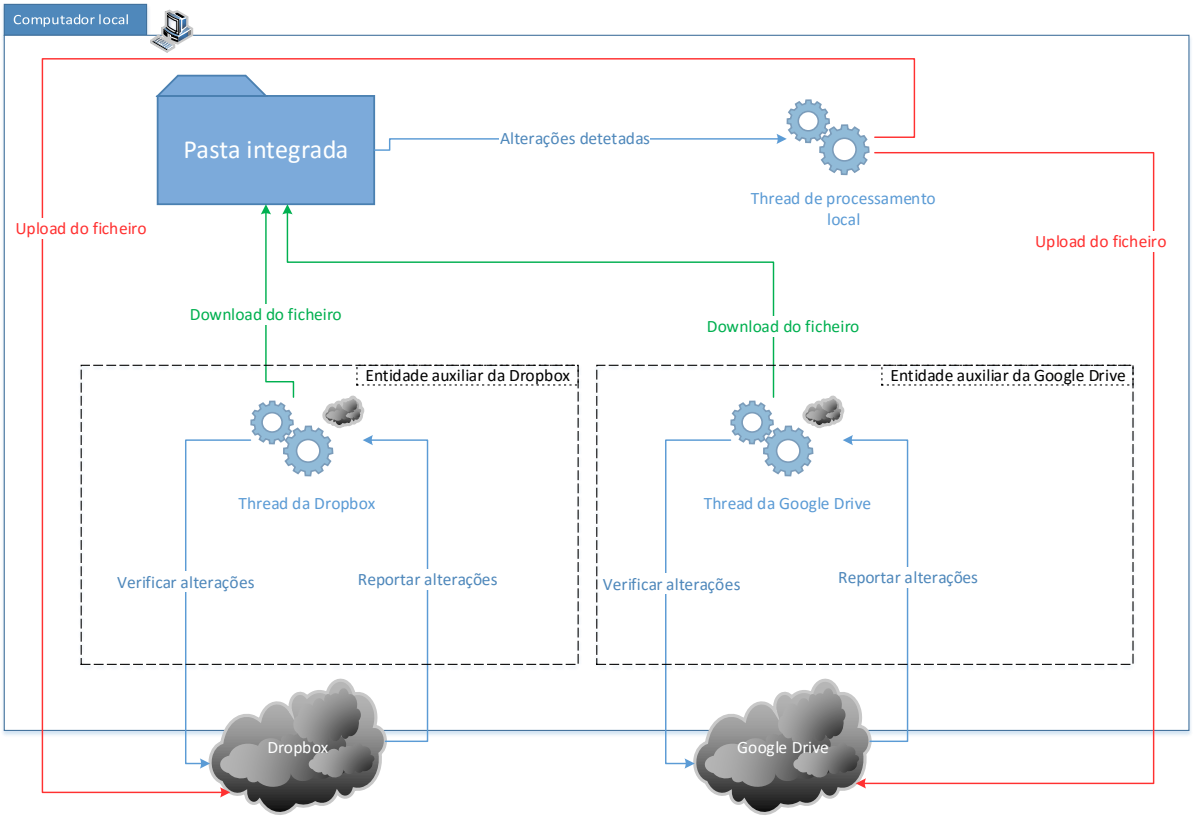


Figura 4.1: Funcionamento e fluxo de execução

4.2 API integrada

Tal como referido anteriormente, foi definida uma API comum para interação com as *clouds*, quaisquer que elas sejam. Esta API será implementada para cada *cloud* usando os métodos por ela disponibilizados. Foram definidos os seguintes métodos:

- `init` - para inicializar a aplicação e efetuar a autenticação, se necessário
- `insert` - para inserir na *cloud* um novo ficheiro criado localmente
- `delete` - para remover da *cloud* um ficheiro apagado localmente
- `update` - para atualizar na *cloud* um ficheiro alterado localmente
- `get` - para obtenção de um ficheiro específico da *cloud*

- `onRemoteChange` - método a ser invocada para atualização local, em consequência de alterações de um ficheiro numa *cloud*
- `getRemainingSpace` - para obtenção do espaço disponível na *cloud*
- `getCloudName` - para obtenção do nome da *cloud*
- `startMonitor` - para começar a monitorização
- `stopMonitor` - para parar a monitorização

Os métodos presentes da API integrada serão abordados mais à frente, nas secções correspondentes aos mecanismos que os usam.

Os métodos `startMonitor` e `stopMonitor` são métodos mais gerais chamados logo após a inicialização dos componentes da classe ou durante a execução do programa. São os métodos que inicializam e interrompem a sincronização, respetivamente.

Para auxiliar a implementação destes métodos na comunicação com as respetivas *clouds*, são utilizadas as suas APIs oficiais. A Dropbox usa a versão `2.0-beta-1` e a Google Drive a versão `v2-rev194-1.21.0`.

4.3 Autenticação perante os serviços *cloud* para autorização de acesso

No capítulo 2, foi referida a distinção entre autenticação e autorização. No entanto, a autorização provoca, no seu primeiro passo, a autenticação perante os serviços. Neste caso, como se pressupõe um único utilizador por conta de cada serviço *cloud*, os termos autenticação e autorização são utilizados de forma a significarem o mesmo.

O acesso a serviços *cloud*, geralmente, é controlado por mecanismos de autenticação, de modo a garantir que apenas os detentores das devidas credenciais os utilizam. De forma a evitar, sempre que se pretende aceder aos serviços, a constante inserção das credenciais e o fornecimento de dados sensíveis (pois as credenciais dão acesso total a todas as funcionalidades) o acesso pode ser controlado por *tokens*. O utilizador comunica uma vez com os servidores de autenticação do serviço a aceder, fornecendo as suas credenciais (por exemplo, um nome de utilizador e uma password), e o servidor fornece-lhe um *token* de autorização para os acessos seguintes. Este *token* poderá ter um prazo de validade. O *token* é guardado localmente e usado para controlo dos acessos seguintes, sem necessidade de re-inserção das credenciais nos servidores de autenticação do serviço.

O protocolo utilizado na grande maioria destes serviços é o OAuth 2.0, pois é considerado por muitos um standard de autorização web. É esse o protocolo usado para a autorização das aplicações terceiras perante a Dropbox e Google Drive.

Compete ao método `init` da API integrada tratar de obter o *token* a partir do servidor do respetivo serviço. A sua implementação nos dois servidores usados é descrita nas secções seguintes.

4.3.1 Dropbox

No caso da Dropbox, o *token* de acesso é obtido através de um código que, por sua vez, é obtido através de um processo de autenticação usando as credenciais do utilizador. A API fornece um endereço web, usado para autenticação e pedido de consentimento de acesso às funcionalidades necessárias. Passando a autenticação e autorizado o acesso às funcionalidades requisitadas, é disponibilizado um código que deve ser fornecido ao programa para ele, por sua vez, o usar para obter o *token*. O *token* é guardado e usado nos acessos subsequentes. Repare-se que as credenciais não são fornecidas ao programa mas sim ao website da Dropbox aquando a aquisição do código, pelo que o programa não tem conhecimento dos dados de conta, dispondo apenas de um *token* que lhe permite aceder às funcionalidades requisitadas.

Para a realização deste fluxo de autenticação, a aplicação precisa de ser identificada por uma `APP_KEY` e uma `APP_SECRET` (obtidos através da consola de desenvolvimento da Dropbox). O conhecimento destes dois dados por si só não permite o acesso a nenhuma *cloud* de nenhum utilizador. O propósito destes dois dados é simplesmente identificar a aplicação que requer autorização sobre a conta do utilizador do serviço *cloud* como não maliciosa e devidamente permitida na consola de desenvolvimento da Dropbox. A `APP_KEY` e a `APP_SECRET` do programa desenvolvido estão embutidas no programa.

As figuras seguintes (4.2 a 4.6) ilustram o processo de obtenção do *token*, realizado pelo programa aquando a sua primeira execução. Inicialmente, o *browser* definido por omissão abre a página (figura 4.2) de obtenção do código.

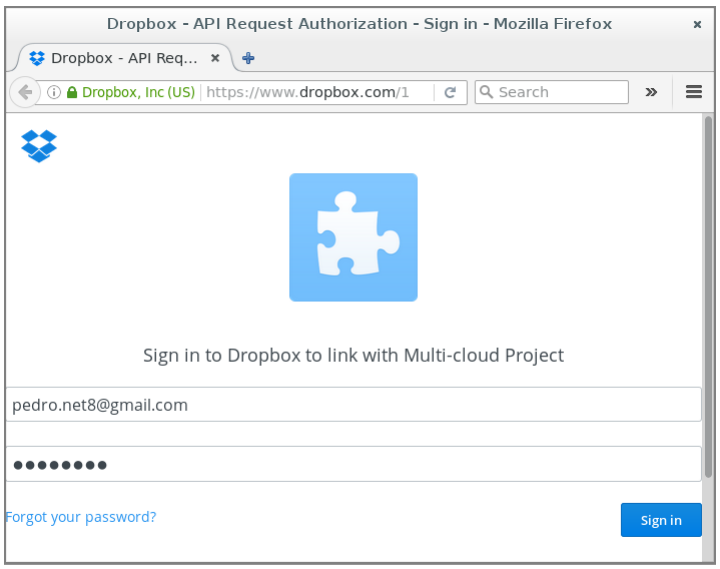


Figura 4.2: Website da Dropbox para obtenção do código

A aplicação abre também uma caixa de diálogo onde se requisita a introdução do código que venha a ser obtido no website (figura 4.3).

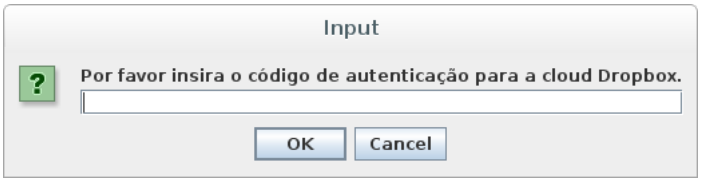


Figura 4.3: Caixa de diálogo para introdução do código, fornecido pela Dropbox, para obtenção do *token* de acesso

No website, o utilizador introduz as suas credenciais (figura 4.2) e, após boa autenticação, é-lhe requisitada autorização para o programa que pretende acesso, listando o nome do programa e quais as funcionalidades a que pretende aceder (figura 4.4).

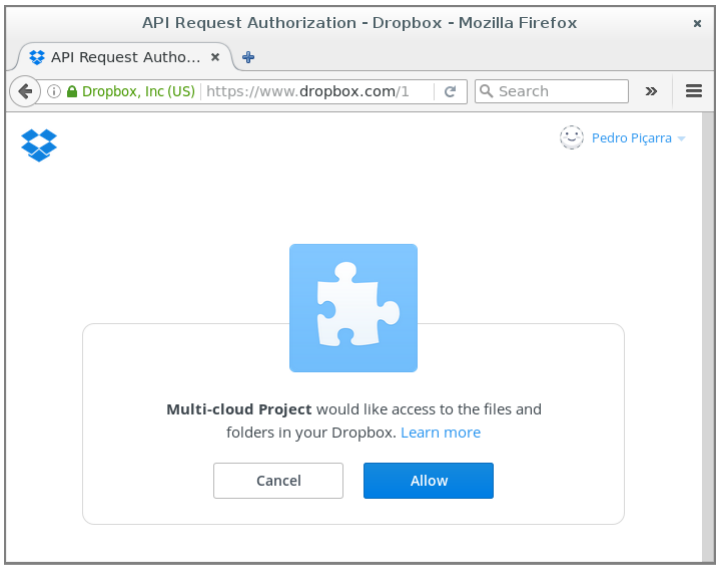


Figura 4.4: Pedido de autorização da aplicação para acesso aos ficheiros da Dropbox

Depois de ser autorizado, é fornecido o código para o programa poder obter o *token* de acesso com o mesmo (figura 4.5).

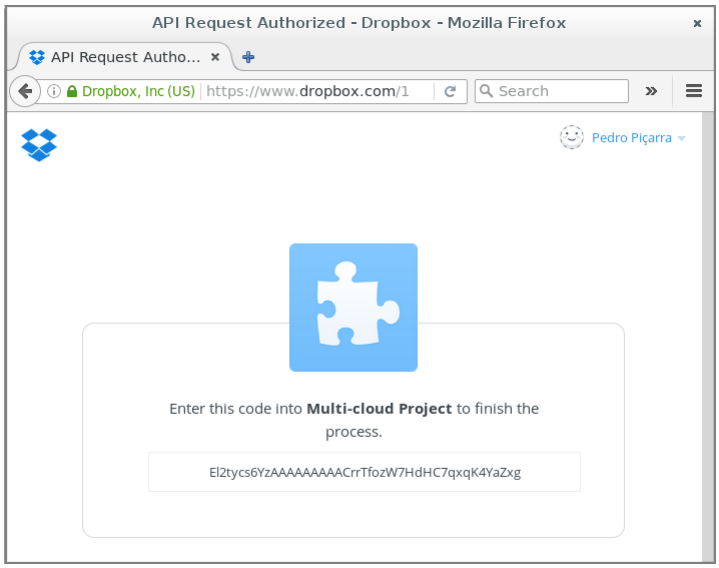


Figura 4.5: Autorização bem sucedida e fornecimento do código para obtenção do *token* de acesso (Dropbox)

Após a introdução do código na caixa de diálogo (figura 4.6), o programa obtém o *token* de acesso e segue a sua normal execução, caso o código esteja correto.

Figura 4.6: Introdução do código, fornecido pela Dropbox, para obtenção do *token* de acesso, no programa

O *token* de acesso fornecido não tem data de expiração e a obtenção de um novo *token* não revoga os anteriores.

4.3.2 Google Drive

No caso do Google Drive, o processo é similar ao da Dropbox. A API existente também disponibiliza um endereço web para obtenção de um código. O código é obtido após boa autenticação, através das credenciais de acesso do utilizador. O código permitirá, analogamente ao processo no caso da Dropbox, obter o *token* de acesso, assim que fornecido à aplicação.

A `APP_KEY` e a `APP_SECRET` são, desta vez, disponibilizadas via ficheiro JSON, obtido previamente na consola de desenvolvimento. Este ficheiro é lido posteriormente no programa, no início da autenticação.

Novamente, é aberto o website, no *browser* definido por omissão, para obtenção do código, exigindo-se credenciais de acesso (figura 4.7).

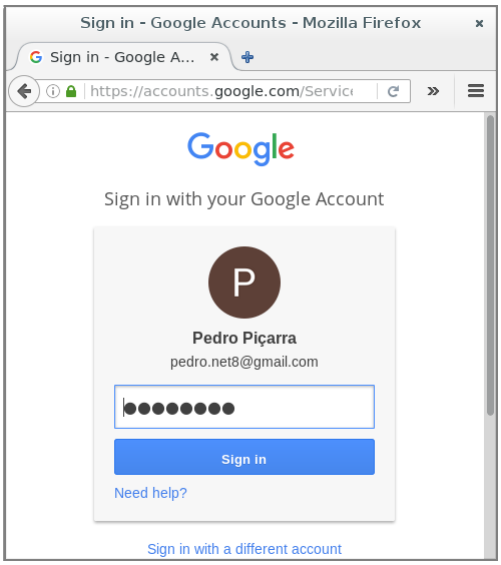


Figura 4.7: Website da Google Drive para obtenção do código

Perante uma autenticação bem sucedida, é requisitada autorização para acesso, por parte da aplicação, aos ficheiros alojados na Google Drive associada à conta autenticada (figura 4.8).

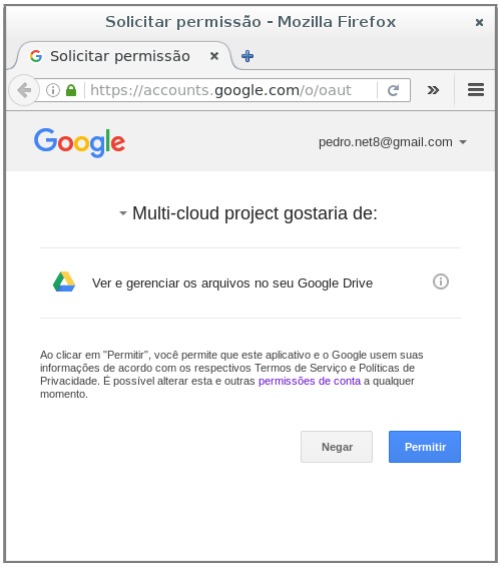


Figura 4.8: Pedido de autorização da aplicação para acesso aos ficheiros da Google Drive

No caso da Google Drive, não é necessário copiar o código para o programa, uma vez que as bibliotecas do Google disponibilizam meios que comunicam diretamente com o *browser*. Após autorização com sucesso e obtenção do código (figura 4.9), o programa recebe o *token* de acesso e usa-o posteriormente para todas as operações sobre ficheiros na *cloud*, seguindo normal execução.[9]

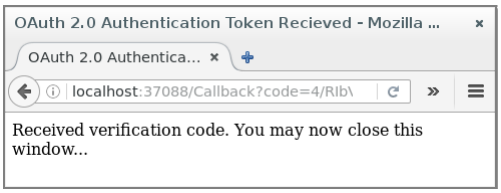


Figura 4.9: Autorização bem sucedida e fornecimento do código para obtenção do *token* de acesso (Google Drive)

4.4 Monitorização de alterações locais

Como referido na Estrutura Geral (4.1), o Java possui suporte para a monitorização local de ficheiros. Foi escolhido este método de monitorização de ficheiros porque, para além de ter um bom desempenho (uma vez que usa métodos nativos do sistema operativo) e de ser altamente compatível com diferentes sistemas (uma vez que se ajusta mesmo quando não tem métodos nativos para utilizar), usa bibliotecas nativas do Java, o que evita o uso de bibliotecas externas que podem não ser tão fidedignas ou otimizadas.

Os métodos são, na sua grande maioria, não bloqueantes, ou seja, quando há um método de leitura, são lidos apenas os dados que estão disponíveis ou nenhuns (no caso de não haverem dados disponíveis de todo). Ao contrário dos métodos bloqueantes onde a thread que os executa fica bloqueada à espera de ter dados que possa ler, nos métodos não bloqueantes a thread pode prosseguir a sua execução mesmo não lendo dados. Da mesma forma, em métodos de escrita, contrariamente aos métodos bloqueantes, nos métodos não bloqueantes a thread pode avançar sem que se tenha acabado de escrever os dados.[13] Isto significa que, no caso dos métodos não bloqueantes, não é estritamente necessário colocar uma thread dedicada à execução desses métodos de leitura e escrita. Significa ainda que podemos tirar partido de mais desempenho, no caso de termos dados disponíveis para leitura ou escrita, pois não temos de esperar até essas operações serem concluídas e, por isso, realizar outras operações entretanto. Porém, se não tivermos dados para ler ou escrever, a execução destes métodos é mais dispendiosa, tendo mais *overhead* que os tradicionais métodos bloqueantes.

No contexto da monitorização de ficheiros, poderá colocar-se a questão do impacto causado pelo prosseguimento imediato de execução do programa, quando um ficheiro é criado, modificado ou apagado. No caso da criação e modificação, é preciso ter o cuidado de verificar se o ficheiro está ainda a ser alterado por outros processos, pois respetivo acontecimento é detetado no início dos mesmos e as operações de envio só podem ser executadas sobre o ficheiro quando este estiver completamente criado ou alterado, respetivamente. No caso do apagamento, não existe qualquer problema, pois a operação respetiva no lado remoto não exige a leitura do ficheiro.

Com o pacote NIO do Java podemos monitorizar pastas à escuta de eventos sobre os ficheiros ou pastas nela contidos. Há três tipos de eventos que podem acontecer:

- `ENTRY_CREATE` - criação de um ficheiro (cabeçalho)

- `ENTRY_MODIFY` - modificação de um ficheiro
- `ENTRY_DELETE` - remoção de um ficheiro

Esses eventos são colocados numa fila, onde são atendidos por ordem de chegada, processados de acordo com o evento.[20] Cada evento, tem um método associado - uma *callback*. No programa implementado, é verificado num ciclo quais os eventos que estão na fila e executado o método conforme o evento.

Existe também uma lista de exceções locais, caso algum evento deva ser ignorado de modo a evitar a circularidade mencionada no capítulo 3. Por exemplo, quando se cria um ficheiro localmente, são acionados dois eventos pelo Java NIO (`ENTRY_CREATE` e `ENTRY_MODIFY`, na criação do cabeçalho do ficheiro e na criação de dados do mesmo, respetivamente). O evento de modificação é acionado devido ao evento de criação do ficheiro (após criação do cabeçalho, quando é adicionado conteúdo ao campo de dados), não sendo necessário executar o método de reação a uma modificação. Como cada evento tem uma *callback* e o evento de criação vai ser tratado na *callback* de atendimento desse mesmo evento associado, podemos ignorar a *callback* de modificação, uma vez que, neste caso, não houve uma modificação no ficheiro.

Por esse mesmo motivo, após uma operação sobre o ficheiro que saibamos que gerará um ou mais eventos a ser ignorados, é adicionada à lista de exceções o caminho do ficheiro, para que possa ser ignorado o próximo evento sobre esse ficheiro, quando for processado, uma vez que os eventos são colocados por ordem de chegada e processados na mesma ordem.

Da mesma forma, à lista de exceções, para além dos casos descritos acima, também são adicionadas exceções sempre que fazemos *download* de um ficheiro, pois quando o descarregamos e o colocamos na pasta sincronizada, vai haver uma notificação do sistema de ficheiros local a reportar que há ficheiros novos ou modificados. No entanto, como esse evento é em resposta ao descarregamento direto dos ficheiros sem alteração de nenhuma outra entidade local, não é necessário enviar para a(s) *cloud(s)* de novo, evitando a circularidade.

Este passo (verificação de exceções) é o primeiro passo a ser executado quando iniciamos o processo de eventos, seguido do processamento dos mesmos. Caso haja uma exceção, remove-se esse evento da lista e saltamos esse ciclo de processamento, ignorando o evento. Caso contrário, verifica-se qual é o evento em questão e executa-se a *callback* específica.

As *callbacks* implementadas são denominadas de acordo com o evento - `OnCreate`, `OnModify` e `OnDelete` em resposta aos eventos `ENTRY_CREATE`, `ENTRY_MODIFY` e `ENTRY_DELETE`, respetivamente:

- `ENTRY_CREATE` →`OnCreate`
- `ENTRY_MODIFY` →`OnModify`
- `ENTRY_DELETE` →`OnDelete`

As *callbacks* dos eventos vão pré-processar o ficheiro associado ao evento (dividir o ficheiro caso necessário e decidir em que *cloud(s)* irá(ão) ser alojado(s)). Após esse pré-processamento, a *callback* executa ainda o *upload* do ficheiro local criado/alterado ou a eliminação do mesmo através dos métodos da API integrada implementados para uma ou mais *clouds*.

Com esta API temos um ponto centralizado onde executamos as operações sobre uma ou mais *clouds*. No caso do processamento dos eventos gerados pela monitorização local, os métodos em foco são o `insert`, `update` e `delete` da API integrada, para o envio dos dados por parte das *callbacks* `OnCreate`, `OnModify` e `OnDelete`, respetivamente:

- `ENTRY_CREATE` →`OnCreate` →`insert`
- `ENTRY_MODIFY` →`OnModify` →`update`
- `ENTRY_DELETE` →`OnDelete` →`delete`

Resumindo, quando um dos eventos locais é acionado, é executada a *callback* para pré-processamento, seguido dos métodos específicos da API integrada, para uma ou mais *clouds*, consoante o evento e características do ficheiro.

A API integrada permite várias implementações da mesma de forma modular. Isto significa que, para cada *cloud*, há uma implementação distinta e, por isso, podemos adicionar outros serviços de armazenamento em *cloud* sem interferir com os que já existem. Significa também que as *callbacks* podem executar os métodos de uma ou mais *clouds*, como evidenciado na figura 4.10.

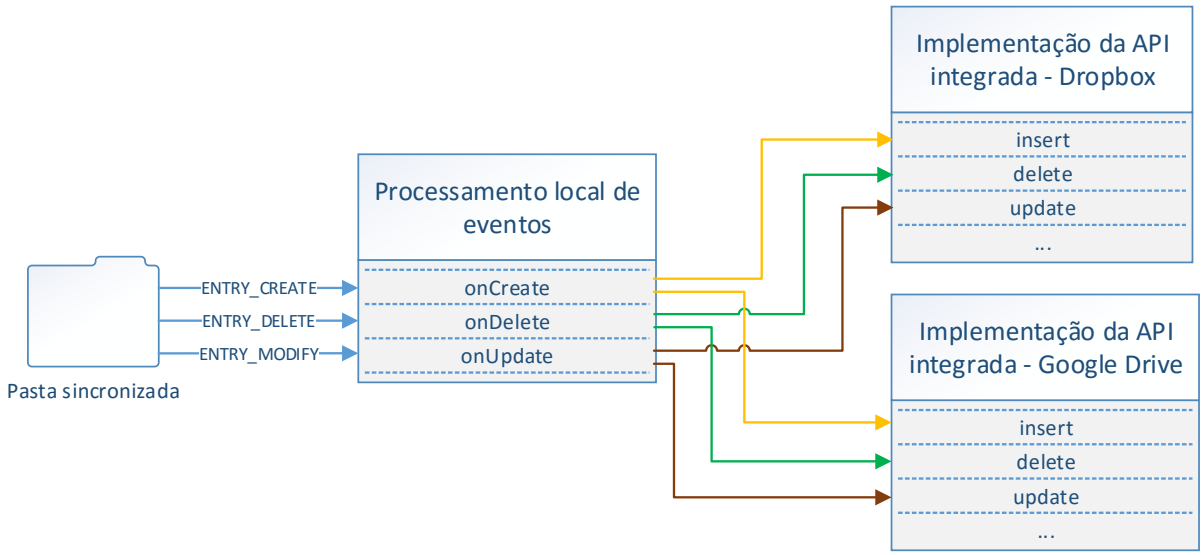


Figura 4.10: Fluxo de execução desde a deteção do evento até à execução dos métodos da API integrada, para uma ou mais *clouds*

Política de distribuição pelas *clouds*

Sempre que localmente é criado um ficheiro novo, é necessário decidir-se em que *cloud* ou *clouds* o alojar. A decisão baseia-se no espaço livre em cada *cloud* no momento da criação. Optou-se por escolher a *cloud* que disponibilize mais espaço. Desta forma, pretende-se distribuir mais uniformemente o espaço livre.

4.5 Monitorização de alterações remotas

Tal como referido na Estrutura Geral (4.1), a monitorização remota assenta no uso de threads, uma por *cloud*. Cada thread verifica regularmente, usando as bibliotecas oficiais da *cloud* que representa, se há alterações remotas a ficheiros. No caso das duas *clouds* implementadas, esta verificação é feita através de *polling* ao servidor a cada segundo. Porém, a forma de o fazer depende das APIs disponibilizadas pelo serviço. Em alternativa aos métodos de *polling*, poder-se-iam usar, por exemplo, métodos bloqueantes de escuta (notificação por eventos).

Caso hajam alterações, o serviço *cloud* correspondente responde com os metadados dos ficheiros alterados e, localmente, é invocado o método `onRemoteChange`. A sua função é fazer refletir localmente as alterações comunicadas pela *cloud*. Essas alterações vão manifestar-se na criação, remoção ou alteração de ficheiros localmente. Há, no entanto, situações especiais a considerar.

Se a alteração acontece num ficheiro especial e reservado para o funcionamento do programa, como por exemplo a pasta de trabalho ou a lista de ficheiros fragmentados (ver secção 4.6), nada é feito. Isto acontece, pois o *upload* destes é feito forçadamente, sempre que é necessário, em vez de confiar na monitorização local e respetivo processamento, pois a presença destes ficheiros, remotamente, é necessária para o bom fluxo da própria monitorização remota, antes de ocorrer a mesma. Ou seja, quando estes ficheiros são alterados, as suas alterações têm de ser refletidas antes das alterações dos ficheiros que afetam.

Se as alterações recebidas dizem respeito a um ficheiro fragmentado, espera-se pela confirmação da existência de todos os fragmentos nas *clouds* (ver secção 4.6) seguido da obtenção dos mesmos, no caso de criação ou modificação. Após obtenção de todos os fragmentos reconstrói-se o ficheiro, sequencialmente. No caso de apagamento, o ficheiro é eliminado localmente.

Se as alterações recebidas pertencem a um ficheiro inteiro, a alteração é refletida diretamente na pasta local, criando-o, modificando-o ou apagando-o conforme o evento.

Finalmente, se as alterações dizem respeito a um ficheiro colocado na lista de exclusões, nada é feito. Existe, para cada thread de monitorização remota, uma lista de exclusões, semelhante à lista de exclusões explicada em 4.4. Com esta lista, tal como a lista de exclusões local, pretende evitar-se a circularidade referida em 3.3. A ela, nas operações sobre ficheiros remotos (métodos `insert`, `update`, `delete`), ou seja, no acto de *upload*, são adicionados os caminhos dos ficheiros a excluir; sempre que se executa um destes métodos é

adicionado o caminho do ficheiro sobre o qual esse método foi executado. Como o ficheiro enviado irá desencadear a notificação de um ficheiro criado, modificado ou apagado no sistema de ficheiros remoto, não queremos agir, localmente, sobre essa alteração remota, pois já foi feito. Por exemplo se adicionarmos um ficheiro (método `insert`) a uma *cloud*, vai haver uma notificação que foi adicionado um ficheiro novo. Porém, queremos que essa notificação seja ignorada, uma vez que não queremos obtê-lo novamente porque acabámos de o inserir.

É de notar que, para cada *cloud*, o programa precisa de analisar que operações sobre o sistema de ficheiros local são necessárias executar aquando uma alteração, pois o serviço *cloud* não nos informa das alterações detalhadamente. Por exemplo, é preciso detetar se um ficheiro foi movido. No caso da Dropbox, esse ficheiro é apagado de uma pasta e criado novamente noutra, pelo que o nosso programa segue este fluxo de execução. Porém na Google Drive, cada vez que um ficheiro muda de localização, só os metadados do ficheiro mudam (neste caso a lista de pastas mãe), pois o sistema de ficheiros remoto da Google Drive não tem a noção de caminho de diretórios, como referido e explicado em 2.1.2.

4.6 Divisão de ficheiros

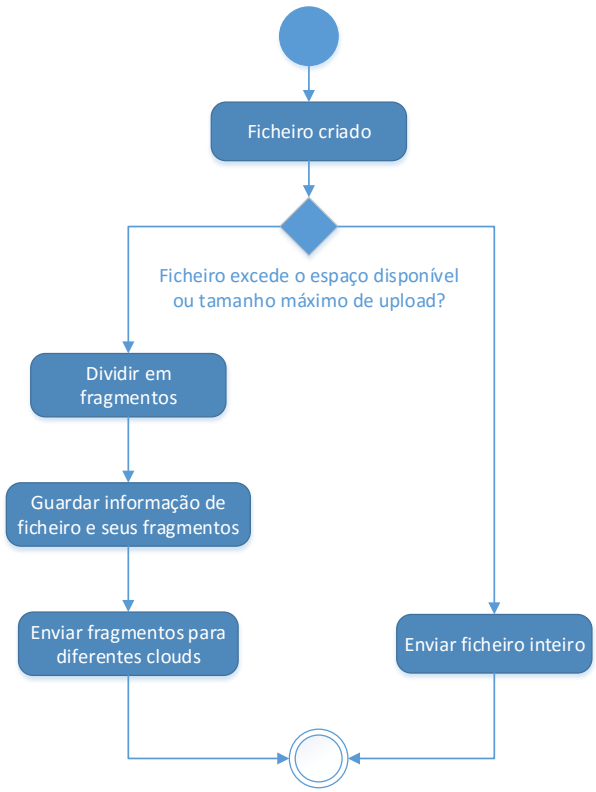


Figura 4.11: Tratamento de ficheiros de tamanho superior ao permitido numa *cloud*

Como descrito em 3.7, há duas situações que nos impedem de colocar um ou mais ficheiros na *cloud* devido ao seu tamanho. Neste caso, é da responsabilidade da *callback* que reage à criação de ficheiros verificar esta situação. Quando um ficheiro é criado ou modificado, antes de ser inserido, é verificado o seu tamanho para saber se é possível ser alocado remotamente. No caso de o tamanho exceder um dos parâmetros descritos em 3.7, o ficheiro é dividido em chunks de tamanho pré-definido. Os parâmetros de tamanho são calculados no início da execução do programa com informações que a API de cada serviço *cloud* específico fornece sobre a conta associada.

Existe, precisamente para estas ocorrências, uma pasta de trabalho chamada `.workDir`. Esta pasta destina-se a trabalhar os ficheiros que precisam de ser divididos ou fundidos e o seu nome é reservado pelo que não podem haver ficheiros ou pastas criados com esse mesmo nome. Todos os ficheiros desta pasta são excluídos da sincronização.

Este processo cria, primeiramente, na pasta de trabalho, uma cópia do ficheiro sobre a qual vamos trabalhar. Depois, o ficheiro é lido sequencialmente e dividido sempre que for lido o tamanho pré-definido do chunk. É importante referir que a razão de gerarmos ficheiros novos, ocupando mais espaço em disco e provocando operações de I/O que influenciam a performance do programa deve-se ao facto de, como referido em 2.1.2, a API da Google Drive não receber como argumento uma stream de dados mas sim os metadados de um ficheiro. Quer isto dizer que, no caso da Google Drive, é a API da própria que cria a stream do ficheiro a ser enviado, não podendo, deste modo, ser executado o *upload* de uma stream controlada pelo programa.

São assim criadas vários fragmentos do ficheiro original. Estes fragmentos são colocados remotamente numa pasta com o nome exato do ficheiro original. Isto previne que se crie um ficheiro de nome igual na *cloud* e fornece um espaço onde se pode guardar as partes de modo a que estejam diretamente organizadas por ficheiro associado. As partes são colocadas em sequência em cada uma das *clouds*, rodando entre elas na mesma ordem. No entanto, o ficheiro original reside na localização primordial da pasta local, o que faz com que este processo seja todo ele transparente para o utilizador.

No decorrer do processo, existe ainda um ficheiro denominado `.splittedFiles` na pasta de trabalho que possui uma lista de todos os ficheiros fragmentados, assim como o número dos seus fragmentos — a base de dados de ficheiros fragmentados. Este ficheiro é uma lista serializada e é essencial para distinguir os ficheiros que devem ou não ser sincronizados e de que forma, uma vez que não se quer obter, no armazenamento local, os ficheiros fragmentados separados. Este ficheiro é ainda crucial para a sincronização de operações entre as *clouds* remotas e o sistema local pois numa situação em que mais que um computador corre o mesmo programa associado às mesmas contas, é fulcral que todos estejam sincronizados para não haver corrupção de informação ou informação duplicada ciclicamente.

Para a implementação da fragmentação de ficheiros foram consideradas 3 soluções. A primeira consiste em fazer atualizar a base de dados de ficheiros fragmentados antes de se fazer *upload* dos fragmentos de forma a que o programa saiba, antes de detetar alterações de ficheiros, que há ficheiros fragmentados. Na figura 4.12, está representado o funcionamento dessa solução quando dois computadores têm a aplicação em funcionamento.

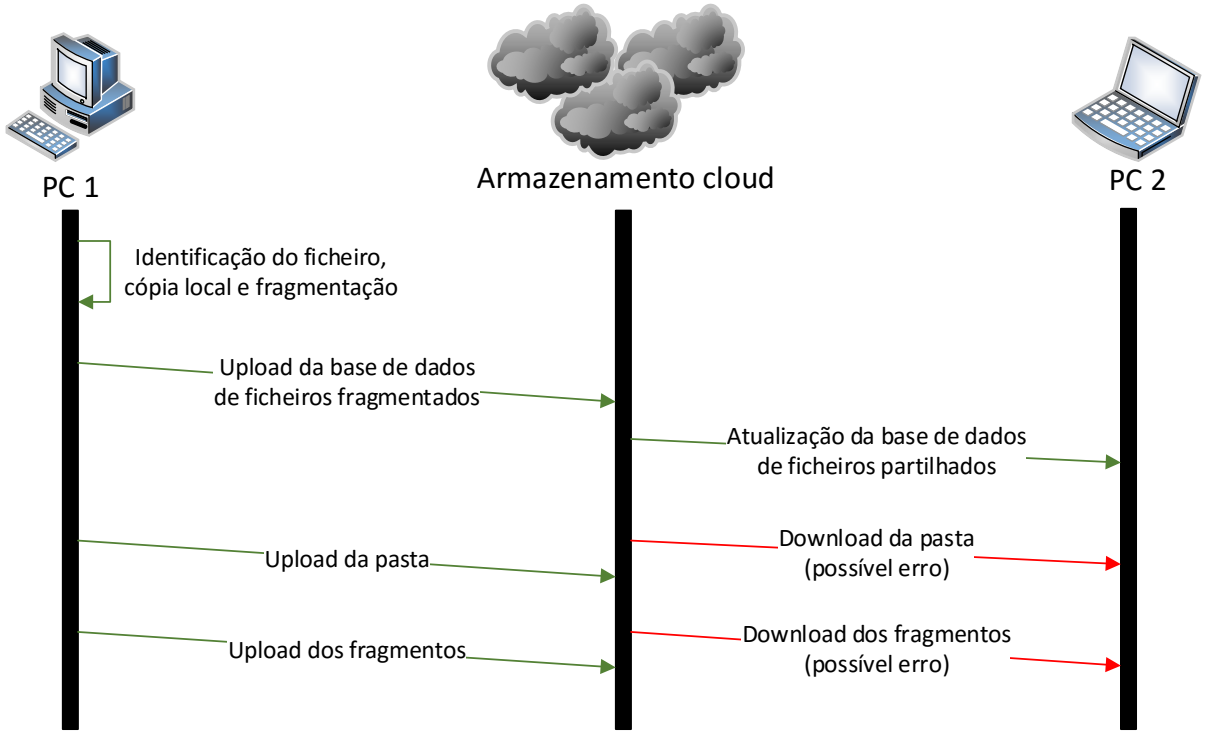


Figura 4.12: Solução de divisão 1 - atualização prévia da base de dados de ficheiros fragmentados. As operações problemáticas encontram-se representadas com uma seta vermelha

---

Infere-se claramente que, deste modo, o programa não sabe quando é que a transferência dos fragmentos está concluída. Por esse mesmo motivo, poderá tentar descarregar os ficheiros e não conseguir.

Outra solução considerada consiste na atualização da base de dados de ficheiros fragmentados e seu *upload* apenas depois de o envio dos fragmentos estar concluído. Na figura 4.13, está representado o funcionamento da respetiva solução quando dois computadores têm a aplicação em funcionamento. Percebe-se que o único problema desta solução é a reação do programa ao ver ficheiros atualizados remotamente sem saber que são fragmentos. Desta forma, estes ficheiros vão ser transferidos como se fossem ficheiros inteiros, provocando um processo de não transparente para o utilizador.

Repare-se então que ao introduzir a pasta com os fragmentos do ficheiro sem qualquer informação para a *cloud* teria como consequência a não distinção deste tipo de ficheiro, o que resultaria no *download* não desejado dos fragmentos. Por outro lado, dar de imediato a informação completa na base de dados de ficheiros fragmentados faria com que o sistema considerasse que existiria, a partir daquele momento, um ficheiro (cujo nome seria o nome da pasta) que teria sido fragmentado e os seus fragmentos (cujos identificadores estariam na base de dados) se encontravam todos presentes dentro dessa pasta. Como o *upload* ainda não tinha sido concluído, ocorreria um erro a tentar obter os fragmentos e a montar o ficheiro.

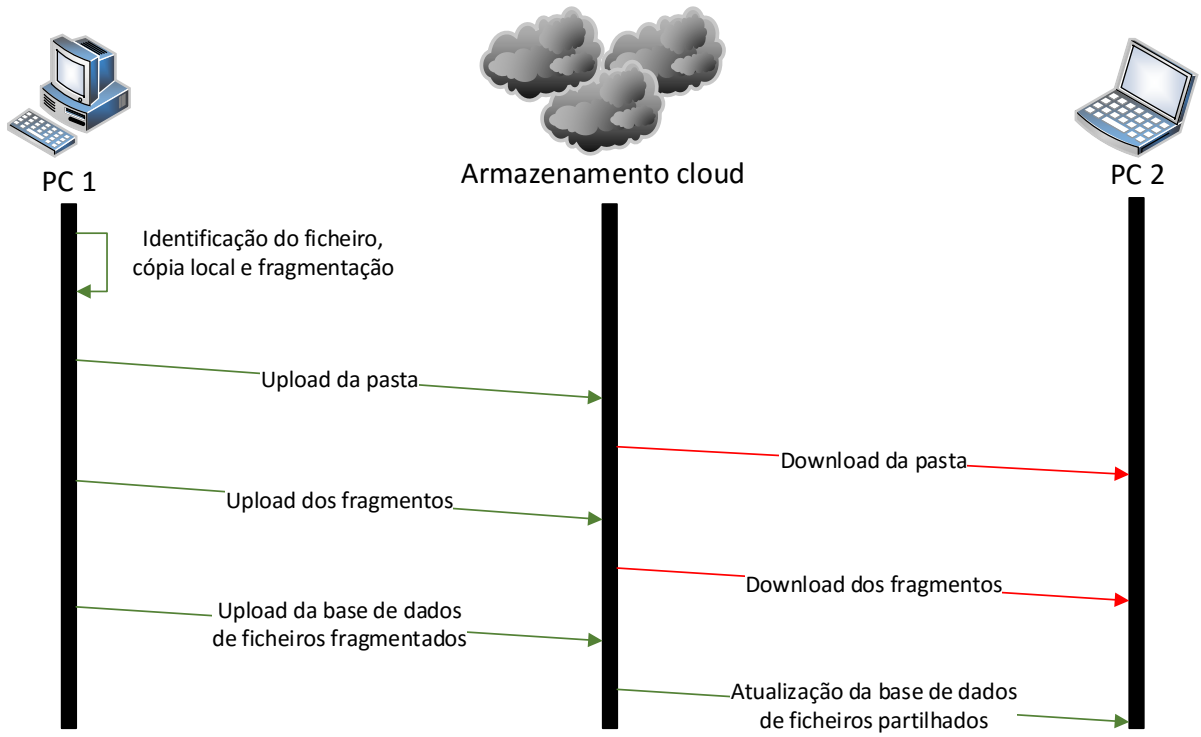


Figura 4.13: Solução de divisão 2 - atualização posterior da base de dados de ficheiros fragmentados

As operações problemáticas encontram-se representadas com uma seta vermelha

---

A última solução consiste na notificação de fragmentação através da base de dados de ficheiros fragmentados em 2 passos:

1. Primeiro, faz-se o *upload* da base de dados de ficheiros fragmentados com a informação do nome do ficheiro, sem fragmentos, imediatamente antes da pasta ser criada, para a thread de verificação de alterações remotas verificar que existirá um ficheiro fragmentado a ignorar de momento (pois ainda não tem registo de fragmentos existentes)

2. A seguir, faz-se o *upload* da mesma base de dados, desta vez com a informação dos fragmentos do ficheiro anterior, imediatamente após os fragmentos serem inseridos para se poder transferir esses mesmos fragmentos e o sistema poder reconstruir o ficheiro sem problemas.

Na figura 4.14 está representada a última solução. Este processo permite a preparação do programa para transferências de ficheiros fragmentados e a transferência de fragmentos apenas quando temos a certeza que todos os envios foram bem sucedidos. Por este motivo, foi a solução adotada para a implementação, sendo a ordem do processo de fragmentação descrito acima o seguinte:

1. Cópia do ficheiro para a diretoria de trabalho

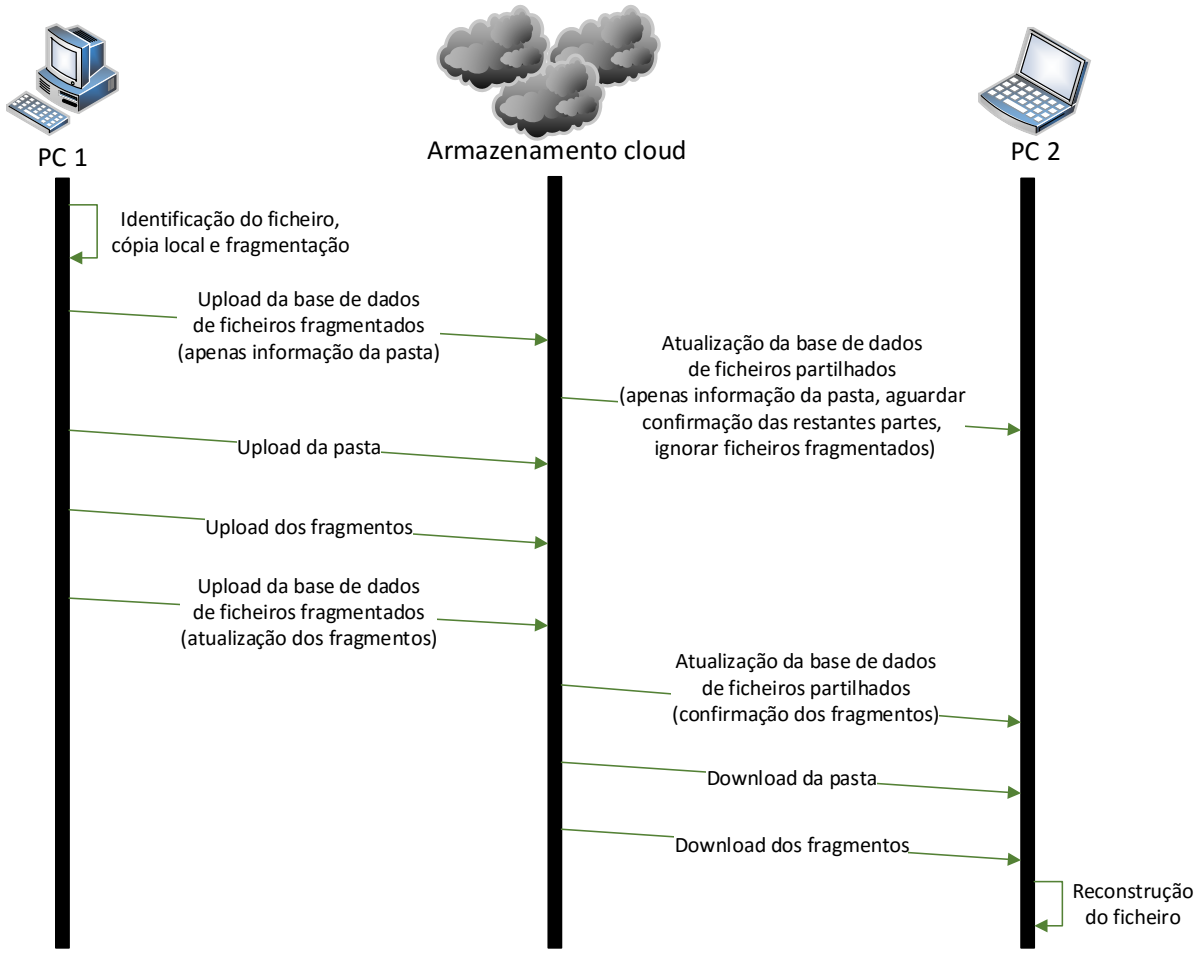


Figura 4.14: Solução de divisão 3 - atualização da base de dados de ficheiros fragmentados em 2 passos

---

2. Entrada na base de dados de ficheiros fragmentados (local)

3. *Upload* da base de dados de ficheiros fragmentados

4. Criação da pasta no armazenamento remoto

5. Fragmentação local

6. *Upload* dos fragmentos para as respetivas *clouds* e, simultaneamente, atualização da base de dados de ficheiros fragmentados (local)

7. *Upload* da base de dados de ficheiros fragmentados

4.7 Base de dados e registo local

A base de dados é crucial para a consistência do programa, pois é necessário sabermos qual o estado dos ficheiros presentes na pasta.

É essencial que a base de dados contenha a informação do ficheiro, da(s) *cloud(s)* a que está associado e outros aspetos importantes como a revisão, o identificador, o *path* associado e o nome do ficheiro.

Os atributos referidos anteriormente têm, cada um, um propósito definido:

- Revisão - Para identificar a versão mais recente do ficheiro presente na pasta sincronizada
- Identificador - Para identificar unicamente um ficheiro no sistema de ficheiros do armazenamento remoto
- Path associado - Para identificar a localização do ficheiro no sistema de ficheiros local
- Nome do ficheiro - Conveniente para operações sobre o ficheiro, nomeadamente organização de pastas.

Foi usado o SQLite para a implementação da base de dados, uma vez que a informação a armazenar não é massivamente grande. O SQLite também é uma ferramenta leve em termos de peso computacional, uma vez que não requer um servidor para o seu funcionamento. Para esta complexidade de dados determinou-se que seria a melhor opção.

A base de dados está organizada em 2 níveis como referido em 3. O primeiro nível indica onde está localizado o ficheiro remotamente, ou seja, tem informação sobre a *cloud* a que está associado. O segundo nível tem informações mais específicas como a revisão, o identificador e o *path* local associado. Cada nível só necessita de uma tabela, uma vez que toda a informação é diretamente associada ao ficheiro em si.

A implementação da base de dados é modular, ou seja, existe um módulo geral para coordenar os diferentes módulos inseridos. Este aspeto permite-nos adicionar várias *clouds* sem interferir com as já existentes no programa. Basicamente, para cada *cloud*, existe uma classe que trata exclusivamente da base de dados. Essa classe respeita uma API e implementa métodos que permitem operações sobre a base de dados:

- addFileRecord - adicionar registo do ficheiro
- getFileRecord - obter registo do ficheiro
- deleteFileRecord - apagar registo do ficheiro
- editFileRecord - modificar registo do ficheiro
- createFileTable - criar tabela de registos
- searchById - procurar por ID do ficheiro, em vez de *path*

Obviamente, o *path* é a chave primária das tabelas tanto no primeiro como no segundo nível. No entanto, nas *clouds*, a chave primária não é o *path* mas sim o ID (um identificador único atribuído na criação do ficheiro), uma vez que os sistemas de ficheiros não se organizam todos da mesma forma.

Por exemplo, a Dropbox consegue obter ficheiros através do *path* local associado, porque faz a conversão desse *path* para o ID que lhe foi atribuído. Já a Google Drive não tem noção de *path* (descrito em 2.1.2. Os ficheiros são armazenados com o identificador único tendo como atributo uma lista de *parents*. Esses *parents* são os identificadores das pastas que contêm esse ficheiro. É necessário que se construa a árvore de ficheiros, percorrendo todas as pastas e listando os ficheiros nelas contidos, atribuindo as hierarquias corretas aos mesmos.

Capítulo 5

Resultados

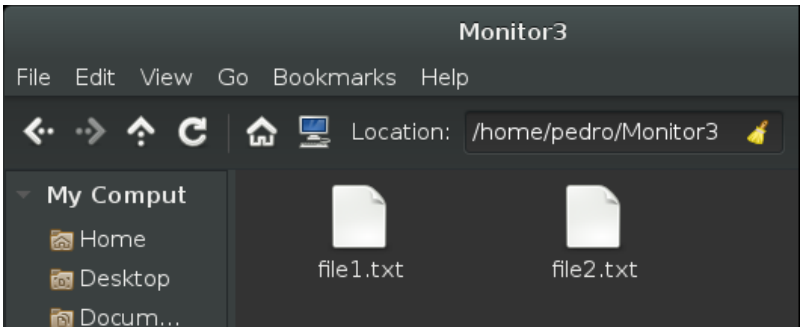
Neste capítulo são apresentadas as experiências realizadas com o objetivo de mostrar e avaliar o funcionamento da solução multi-cloud desenvolvida no âmbito desta dissertação. A primeira experiência mostra a visão integrada e a distribuição de ficheiros pelas duas *clouds* usadas na implementação. A segunda experiência mostra o funcionamento num cenário em que um ficheiro tem de ser fragmentado. Finalmente, a última experiência mostra alguns número sobre o desempenho da solução desenvolvida no contexto dos ficheiros fragmentados.

5.1 Visão integrada e distribuição pelas *clouds*

O objetivo central da aplicação desenvolvida é juntar espaços de armazenamento em duas ou mais *clouds* e disponibilizá-los ao utilizador como um único espaço integrado. Com o propósito de mostrar esta funcionalidade foi realizada a seguinte experiência. A aplicação subscreveu-se a duas *clouds* (Dropbox e Google Drive) e garantiu que o espaço disponibilizado por cada uma era o mesmo. Como o Dropbox disponibiliza 2 GB e o Google Drive disponibiliza 15 GB, a aplicação artificialmente reduziu o segundo para 2 GB.

A seguir, localmente, criam-se dois ficheiros na pasta integrada. De acordo com a funcionalidade definida, a aplicação deveria alojar os dois ficheiros em *clouds* diferentes, porque, após alojamento do primeiro numa das *clouds*, a outra *cloud* ficaria com o espaço livre maior.

A figura 5.1 evidencia este facto. Em cima, mostra-se a pasta local, onde se podem ver os dois ficheiros criados. Na parte de baixo da figura, tem-se uma visão web das duas *clouds*, onde se pode constatar que cada uma contém apenas um dos ficheiros.



(a)

(b)

(c)

Figura 5.1: Ficheiros distribuídos em *clouds* distintas: (a) pasta integrada; (b) alojamento na Dropbox; (c) alojamento na Google Drive.

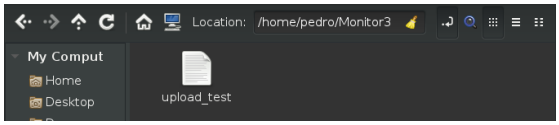
5.2 Divisão de ficheiros

Como referido nos capítulos 3 e 4, o programa contempla um mecanismo de fragmentação de ficheiros cujos tamanhos excedam determinados critérios. Fizeram-se alguns testes para mostrar este mecanismo.

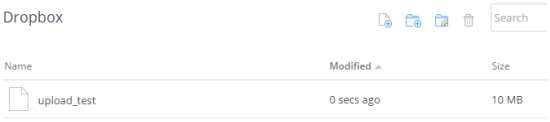
5.2.1 Divisão ao adicionar ficheiros

Para este teste, o programa foi condicionado a usar ficheiros inteiros até 10 MiB. Ficheiros de tamanho igual ou acima de 10 MiB são fragmentados em *chunks* de 2 MiB.

Primeiramente, foi colocado um ficheiro com 10 485 759 Bytes (menos 1 byte de 10 MiB) na pasta integrada. O resultado é o *upload* do ficheiro como um todo. Isto mesmo pode ser constatado na figura 5.2, onde se mostra o conteúdo da pasta local e o seu alojamento na Dropbox.



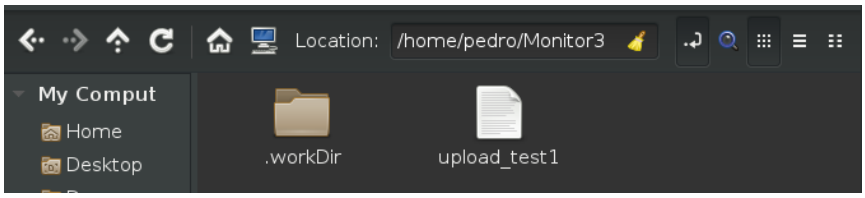
(a)



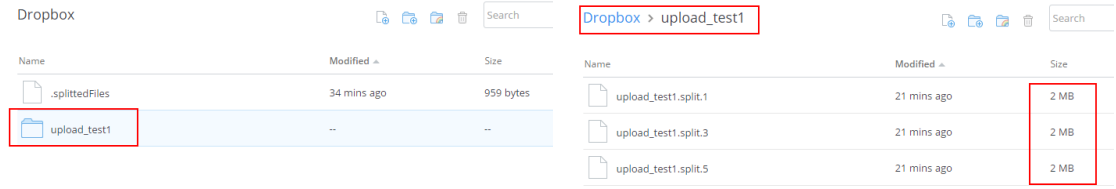
(b)

Figura 5.2: *Upload* de ficheiro sem fragmentação: (a) pasta integrada; (b) alojamento remoto

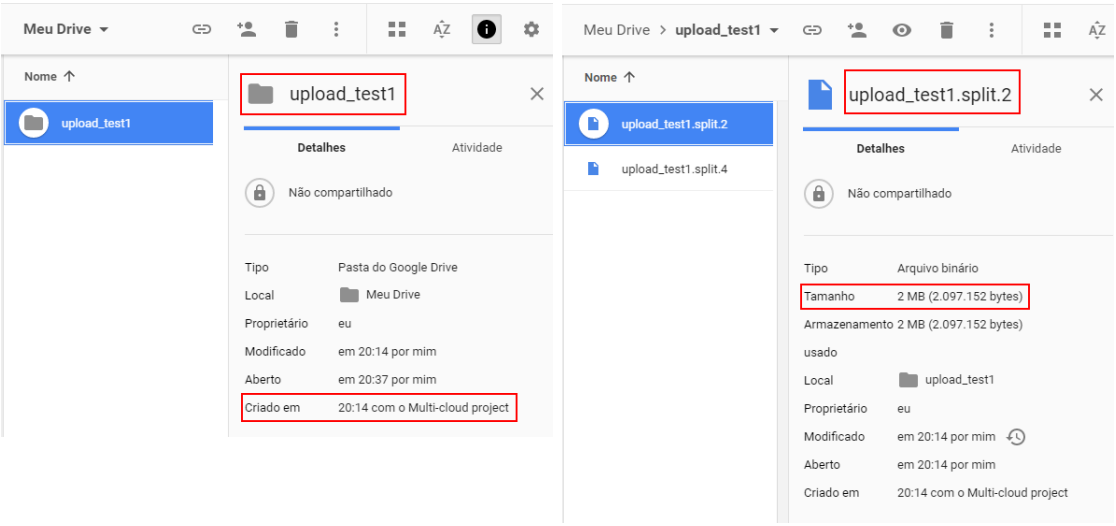
A seguir colocou-se outro ficheiro de 10 MiB na pasta local. O tamanho do ficheiro ultrapassa o tamanho máximo sem fragmentação, por isso, o ficheiro foi dividido e armazenado em diferentes *clouds*, tal como é mostrado pela figura 5.3.



(a)



(b)



(c)

Figura 5.3: *Upload* de ficheiro com fragmentação: (a) pasta integrada; (b) alojamento na Dropbox; (c) alojamento na Google Drive.

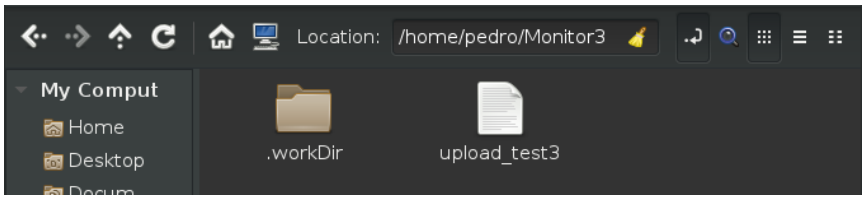
5.2.2 Divisão ao modificar ficheiros

Fez-se um segundo teste em que o ficheiro ultrapassa o limite de fragmentação após uma operação de atualização. Neste teste, o programa foi condicionado a usar ficheiros inteiros até 10 MiB, tal como anteriormente, com a exceção de que os ficheiros só são repartidos quando o seu tamanho é superior 10 MiB, em vez de igual ou superior como feito em 5.2.1.

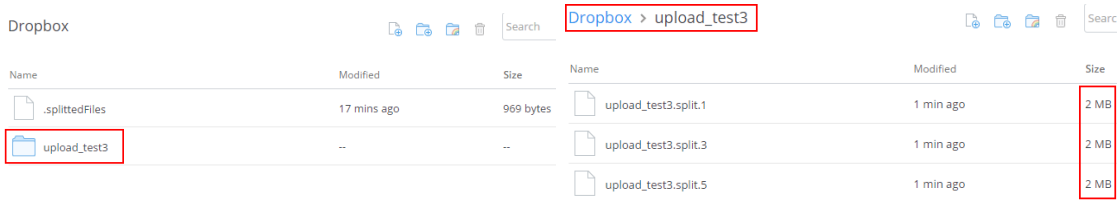
Colocou-se um ficheiro de 10 MiB na pasta partilhada. Verificou-se novamente o *upload* do ficheiro inteiro, da mesma forma que em 5.2.1. Adicionou-se, de seguida, 4 Bytes de conteúdo ao ficheiro, usando o comando

```
echo aaaa >> upload_test3
```

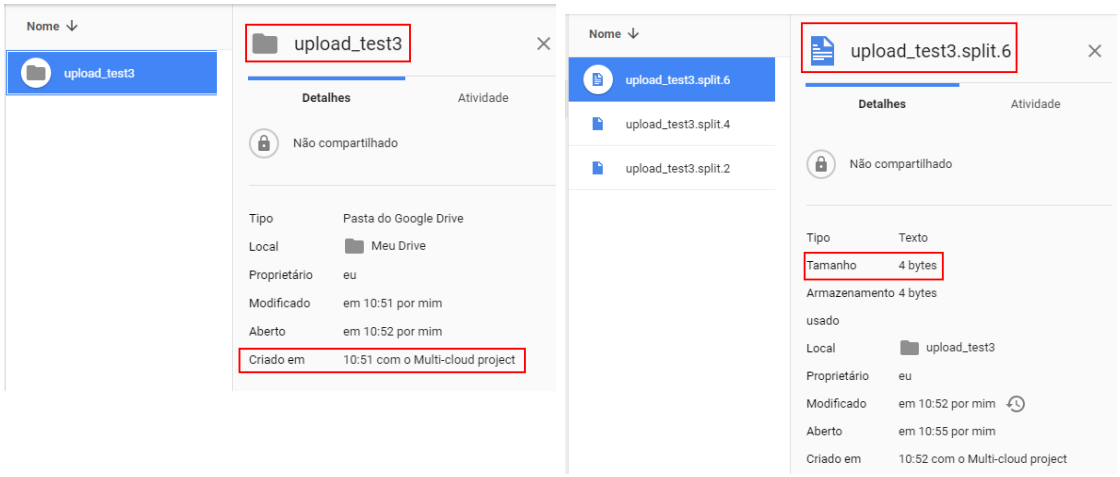
Isto fez com que o ficheiro ultrapassasse o limite de fragmentação. Como o ficheiro originalmente tinha 10 MiB, são criadas 5 partes de 2 MiB e ainda uma última com os 4 Bytes adicionados, como ilustrado na figura 5.4. Portanto, o ficheiro inteiro foi eliminado e o mesmo foi dividido em partes. Da mesma forma, embora não demonstrado, quando o ficheiro é modificado de forma a ter um tamanho menor que o limite de ficheiros inteiros, os fragmentos são eliminados e é feito o *upload* do ficheiro inteiro.



(a)



(b)



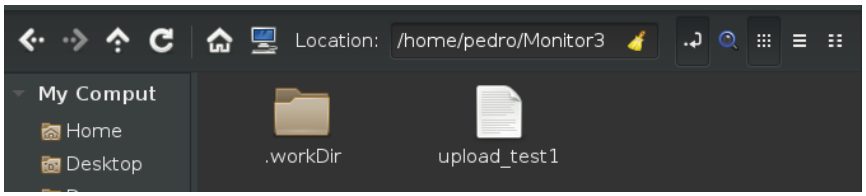
(c)

Figura 5.4: Fragmentação de ficheiros após modificação: (a) pasta integrada; (b) alojamento na Dropbox; (c) alojamento na Google Drive.

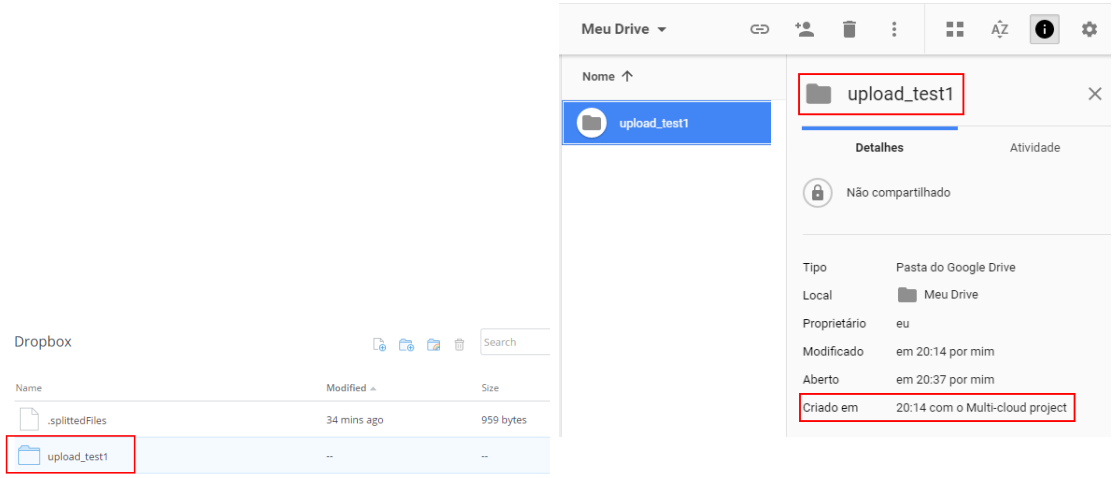
5.2.3 Deteção da fragmentação

Nos testes anteriores mostrou-se o tratamento de ficheiros cujo tamanho exceda o limite de fragmentação e o alojamento dos fragmentos nas *clouds* integradas. Neste teste, mostra-se que as instâncias da aplicação em outros computadores identificam que se trata de um ficheiro fragmentado e criam-no localmente como um todo.

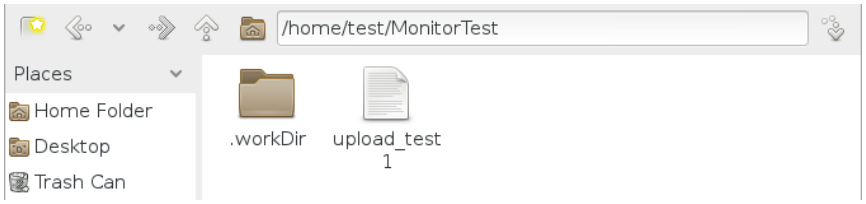
Quando se coloca um ficheiro com tamanho inferior ao limite de fragmentação na pasta local, o ficheiro é enviado para a *cloud* como um todo e descarregado como um todo nas outras máquinas. Se o ficheiro sofreu fragmentação, os fragmentos são descarregados e o ficheiro é reconstruído. Isso é mostrado na figura 5.5, baseada na figura 5.3, à qual se acrescentou a pasta integrada numa segunda máquina.



(a)



(b)



(c)

Figura 5.5: Reconstrução de ficheiro fragmentado: (a) pasta integrada onde ocorreu a fragmentação; (b) alojamento nas *clouds*; (c) pasta integrada onde ocorreu a reconstrução.

5.3 Desempenho

É importante referir que mecanismos como a divisão de ficheiros podem ter grandes impactos em performance. Como tal, também foi considerado essencial ter uma análise desse impacto.

5.3.1 Condições de teste

Os testes de desempenho foram realizados numa máquina virtual com um processador de 4 cores e 2 GB de memória RAM, correndo uma distribuição Linux Mint de 64 bits. O computador anfitrião contém um processador Intel i7 Q720 @1.60GHz, 6 GB de memória RAM, uma placa de rede sem fios integrada e um disco SSD num barramento SATA-II.

Previamente, foram testadas algumas atividades de I/O para verificar se havia diferenças significativas na velocidade de escrita em disco entre o ambiente duma máquina virtual (com *VMware*) e o ambiente nativo. Foi verificado que não existem diferenças significativas, o que era expectável, uma vez que as tecnologias de virtualização estão bastante avançadas e os controladores do disco encontram-se implementados de uma forma bastante eficiente.

Os testes pretendem testar o desempenho do programa numa situação mais próxima do real possível. Uma vez que esta aplicação se destina ao utilizador comum, os testes foram realizados numa habitação com uma rede sem fios com acesso à Internet cujas velocidades máximas são de 60 Mb/s e 10 Mb/s de *download* e *upload*, respetivamente.

5.3.2 Testes

Os testes recolhem os tempos de execução com base no método `nanoTime()` do Java. Primeiramente, para uma dada *cloud*, é feito *upload* de um ficheiro de 10 MiB. Este processo é repetido para o mesmo ficheiro mas desta vez dividido em fragmentos, começando em 2 e aumentando progressivamente até 10.

Cada uma das situações é repetida 10 vezes e são calculados os tempos médios das operações mais relevantes, nomeadamente:

- Inserção de todos os fragmentos
- Inserção de cada fragmento
- Inserção de informações nas bases de dados
- Divisão do ficheiro
- Limpeza de ficheiros

Uma vez que o tempo das operações na Internet são muito variáveis devido a múltiplos fatores externos, é importante ter a informação das oscilações desses mesmos tempos. Por

isso, são ainda calculados o desvio padrão e o erro do tempo medido para um intervalo de confiança de 95%.

No que toca à troca de dados na Internet, optou-se por estudar apenas o fluxo de envio, pois este é o mais limitado na grande maioria das ligações e o causador de maior espera neste tipo de operações.

5.3.3 Custo da divisão de ficheiros

Querendo avaliar-se o custo resultante da divisão de ficheiros, executou-se o teste referido em 5.3.2 usando a Dropbox. O objetivo é avaliar o impacto da divisão de um ficheiro no tempo de *upload*. Os resultados obtidos estão presentes na tabela 5.1.

Tabela 5.1: Desempenho da fragmentação de ficheiros para a Dropbox com 1 thread de upload

		Tempo médio (s)	Desvio padrão (s)	Erro intervalo de confiança de 95% (s)	Tempo por fragmento
Inserção do ficheiro	Inteiro	32,6733	1,42294	0,88195	-
	2 fragmentos	35,49699	1,40482	0,87072	17,74849
	3 fragmentos	36,68789	1,77347	1,09921	12,2293
	4 fragmentos	37,99247	1,67191	1,03626	9,49812
	5 fragmentos	38,28248	1,13252	0,70194	7,6565
	6 fragmentos	39,85249	2,12859	1,31931	6,64208
	7 fragmentos	39,63828	1,30819	0,81083	5,66261
	8 fragmentos	41,21867	1,46688	0,90918	5,15233
	9 fragmentos	43,38646	2,7278	1,69071	4,82072
	10 fragmentos	44,60985	3,2802	2,03309	4,46099
Entrada na base de dados	DB comum	0,21043	0,02889	0,01791	-
	DB ficheiros fragmentados	0,09548	0,26323	0,16315	-
Outros	Divisão	0,43726	0,18209	0,11286	-
	Limpeza	0,00172	0,00081	0,0005	-

Com o aumento do número de fragmentos, nota-se o aumento progressivo do tempo médio de inserção. Este aumento é esperado, uma vez que para além da divisão, o número de conexões TCP aumenta. Como o TCP é um protocolo com um mecanismo *three-way handshake*, no início da conexão existe uma troca de pacotes necessária para inicialização. Cada ligação produz estes pacotes, o que faz com que o número de pacotes a transmitir aumente com o número de fragmentos e, consequentemente, o tempo de transmissão. Com o desvio padrão e o erro para um intervalo de confiança de 95%, conseguimos ainda reter que as oscilações entre tempos não são muito grandes ao aumentarmos o número de fragmentos.

Tabela 5.2: Desempenho da fragmentação de ficheiros para a Google Drive com 1 thread de upload

		Tempo médio (s)	Desvio padrão (s)	Erro intervalo de confiança de 95% (s)	Tempo por fragmento
Inserção do icheiro	Inteiro	29,95543	0,46682	0,28934	-
	2 fragmentos	35,02489	1,79339	1,11155	17,51245
	3 fragmentos	35,06541	1,71291	1,06167	11,68847
	4 fragmentos	35,56252	1,17294	0,69317	8,89063
	5 fragmentos	36,42274	0,79683	0,49388	7,28455
	6 fragmentos	37,29622	1,07008	0,66324	6,21604
	7 fragmentos	39,83957	1,72484	1,06907	5,69137
	8 fragmentos	40,95690	1,39497	0,86461	5,11961
	9 fragmentos	42,64220	1,76383	1,09323	4,73802
	10 fragmentos	43,07447	1,78220	1,10462	2,87163
Entrada na base de dados	DB comum	0,18112	0,03086	0,01913	-
	DB ficheiros fragmentados	0,35058	1,03251	0,63996	-
Outros	Divisão	0,62791	0,77158	0,45598	-
	Limpeza	0,00227	0,00057	0,00034	-

Comparativamente ao desempenho usual de um único ficheiro inteiro, os tempos parecem aumentar cerca de um segundo por cada fragmento adicionado. Este valor de acréscimo depende também, claro, do tamanho do ficheiro, pois estamos a contar com o tempo de divisão do mesmo (cerca de meio segundo, em média) e ainda com o número de acessos às bases de dados, embora este seja desprezável. No entanto, se definido um tamanho justo para os fragmentos, o programa parece ter um desempenho não muito distante da situação original, uma vez que, na maioria das situações, alguns segundos acabam por não ter grande significância para o utilizador.

5.3.4 Comparação entre *clouds*

No capítulo 2 foi referido que a Google Drive tem um método de transporte diferente do da Dropbox. Com o objetivo de verificar se há diferenças significativas entre *clouds*, foram executados os testes referidos acima, em 5.3.2, também para a Google Drive. Os resultados obtidos são apresentados na tabela 5.2.

Consegue perceber-se pelas tabelas 5.1 e 5.2, e pelo gráfico da figura 5.6, que representa os tempos médios em função do número de fragmentos, que a Google Drive tem um desempenho melhor que a Dropbox, em todos os cenários. Porém, a melhoria não é muito significativa e diminui com o número de fragmentos. Não se consegue propriamente dizer que a Google Drive tem uma vantagem sobre a Dropbox neste teste.

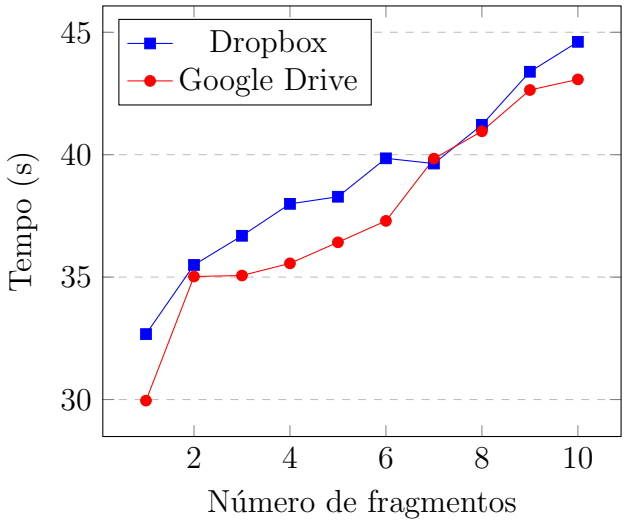


Figura 5.6: Desempenho - Dropbox vs Google Drive

Poder-se-ia apontar a escassa velocidade de *upload* como responsável por esta proximidade. Porém, testadas 80 inserções de um ficheiro de 40MB na rede sem fios da Universidade de Aveiro (com velocidades de *upload* de 1Gb/s), tanto para a Google Drive como para a Dropbox, nota-se uma diferença de cerca de 1 segundo.

Conclui-se, por isso, que não há grande diferença entre a Google Drive e a Dropbox em termos de tempos de transporte de dados. Note-se que este facto pode variar com a localização e até com diferentes momentos do dia devido às diferentes quantidades de tráfego na Internet. Ainda assim, no caso geral, não parece afetar de forma a que seja relevante.

5.3.5 Paralelização

Ainda na fase do desenvolvimento do programa, verificou-se alguma latência quando era necessária a divisão de ficheiros. Naturalmente, era previsível que o ponto de maior latência fosse o *upload*, pois depende não só dos equipamentos locais como também do servidor. Consequentemente, numa tentativa de aumentar a eficiência, a aplicação foi implementada usando várias *threads* para efetuar a operação de *upload*, permitindo que a fase com maior bloqueio fosse executada em paralelo e não impedisse a execução das outras operações menores.

Isto significa que, nos casos em que o ficheiro é fragmentado, cada fragmento pode ser enviado através de um fio de execução paralelo, o que possibilita que as operações de acesso à base de dados e limpeza sejam executadas ao mesmo tempo que o envio de outros fragmentos. Todavia, o tratamento do próximo ficheiro na lista de espera terá de ser feito depois deste processo, aguardando que todas as *threads* de *upload* terminem.

Para avaliar o efeito da paralelização, o teste mencionado em 5.3.2 foi realizado usando uma única *cloud* (a Dropbox), com diferentes números máximos de *threads* de execução.

Tabela 5.3: Desempenho da fragmentação de ficheiros para a Dropbox com 4 threads de upload

		Tempo médio (s)	Desvio padrão (s)	Erro para intervalo de confiança de 95% (s)	Tempo por fragmento
Inserção do ficheiro	Inteiro	31,39345	0,98953	0,61332	-
	2 fragmentos	33,99629	1,06445	0,65975	16,99814
	3 fragmentos	33,12887	1,11981	0,69407	11,04296
	4 fragmentos	33,94503	1,92935	1,19582	8,48626
	5 fragmentos	33,10604	1,65442	1,02542	6,62121
	6 fragmentos	33,69112	1,33194	0,82555	5,61519
	7 fragmentos	34,51149	1,68792	3,52541	4,93021
	8 fragmentos	35,22230	1,65242	3,50340	4,40279
	9 fragmentos	34,04665	2,78301	1,72493	3,78296
	10 fragmentos	33,33552	2,22909	1,38160	2,22237
Entrada na base de dados	DB comum	0,17102	0,03111	0,01821	-
	DB ficheiros fragmentados	0,36134	1,01087	0,63658	-
Outros	Divisão	0,61998	0,75832	0,45328	-
	Limpeza	0,00221	0,00055	0,00031	-

Para além do caso de uma única *thread*, foram consideradas 2 situações: 4 e 10 *threads*. No primeiro caso, 4, tem-se um número de *threads* igual ao número de núcleos do processador usado. No segundo caso, 10, o número de *threads* é superior ao suportado pelo processador. Os resultados deste teste residem nas tabelas 5.3 e 5.4 e estão ainda graficamente representados na figura 5.7.

Tabela 5.4: Desempenho da fragmentação de ficheiros para a Dropbox com 10 threads de upload

		Tempo médio (s)	Desvio padrão (s)	Erro intervalo de confiança de 95% (s)	Tempo por fragmento
Inserção do ficheiro	Inteiro	31,38745	0,97917	0,61332	-
	2 fragmentos	32,99528	1,06208	0,65975	16,49764
	3 fragmentos	33,12764	1,11634	0,69407	11,04255
	4 fragmentos	34,00561	1,93001	1,19582	8,50140
	5 fragmentos	56,25266	24,43457	15,1447	11,25053
	6 fragmentos	60,72037	29,99619	18,59183	10,12006
	7 fragmentos	61,69603	43,31446	26,84658	8,81372
	8 fragmentos	54,31385	19,42426	12,03928	6,78923
	9 fragmentos	73,39121	51,23025	31,75284	8,15458
	10 fragmentos	57,78988	25,88577	16,04417	3,85266
Entrada na base de dados	DB comum	0,17673	0,03365	0,02015	-
	DB ficheiros fragmentados	0,34775	1,00016	0,62546	-
Outros	Divisão	0,63004	0,76254	0,44137	-
	Limpeza	0,00346	0,00045	0,00032	-

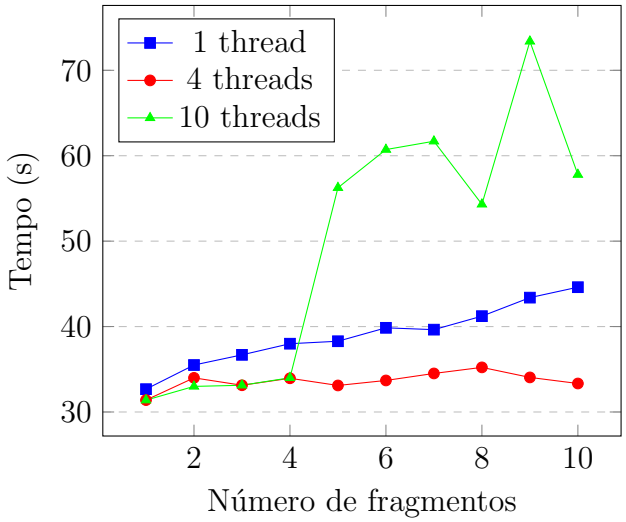


Figura 5.7: Comparação do número de threads de upload - Dropbox

Das tabelas 5.1, 5.3 e 5.4 e ainda da figura 5.7 pode inferir-se que o multi threading poderá reduzir o tempo de execução na fase de *upload*, especialmente com um múltiplo número de fragmentos. Como era de esperar, os tempos de divisão, de acesso às bases de dados e de limpeza não se alteram.

Ao usar-se um máximo de 4 *threads*, o programa parece adaptar-se bem e gerir as conexões de forma eficiente. Os tempos diminuem, não drasticamente claro, pois só se aproveita para executar algumas instruções enquanto as transferências são efetuadas. A velocidade máxima de Internet terá, ainda assim, de ser partilhada pelos 4 fragmentos de cada vez. Esse tempo de processamento extra que foi poupado parece ser maior que o *overhead* provocado pela inicialização das conexões TCP, pois com esta medida, temos tempos totais menores.

Nota-se também que se começa a ter uma maior instabilidade de valores a partir de 3 fragmentos. Na situação de 4 fragmentos, surpreendentemente, tem-se valores de desvio padrão e de erro para um intervalo de confiança de 95% acrescidos. Isto pode dever-se ao facto deste programa não ser o único em execução no sistema. Esta situação é algo a ter em atenção para escolher o número de *threads* de *upload* apropriado. No momento da execução deste programa, outro programa poderá estar a realizar operações e a ocupar tempo de utilização do processador.

Todavia, se se usar um máximo de 10 *threads*, causa-se uma alta instabilidade ao sistema, com tempos bastante superiores quando comparados ao uso de 4 *threads*. Como é de esperar, até 4 fragmentos, o desempenho é semelhante à situação anterior. A partir dos 5 fragmentos, o sistema operativo terá de desempenhar um papel fundamental na gestão do processador, uma vez que não temos recursos de hardware suficientes para gerir 5 *threads*. Para além dos tempos aumentarem intensamente, a sua gama de valores também alarga abruptamente, significando altas oscilações nos seus tempos de inserção. Este facto pode dever-se a algumas razões:

1. O excessivo número máximo de *threads* faz com que, ao alternar entre elas, hajam perdas de pacotes, o que provoca, não só retransmissões como também o decréscimo da janela de fluxo. Uma vez que são muitas conexões em simultâneo, essa janela não pode depois subir rapidamente, provocando velocidades mais baixas e, consequentemente, tempos maiores.

2. Uma vez que há mais pacotes a serem gerados ao mesmo tempo, existe a possibilidade de um congestionamento no processamento dos mesmos por parte do sistema operativo, da placa de rede e do ponto de acesso onde estamos ligados (devido a protocolos NAT, etc)

Outro facto ocorrente, não mencionado anteriormente, e que apoia o que foi referido anteriormente é o número de tentativas falhadas no *upload* de dados. Na situação de 5 a 7 ficheiros, foram precisas 3 tentativas de envio. Já de 8 a 10, foram precisas 5 tentativas.

Conclui-se que é preciso um número máximo de *threads* adequado para atingir um bom desempenho. Poderia pensar-se no limite de *threads* que o hardware pode atender a um dado momento, porém, parece ser prudente considerar também os restantes programas em execução, mesmo estes estando inativos.

Capítulo 6

Conclusão e trabalho futuro

No âmbito desta dissertação projetou-se e desenvolveu-se uma solução *desktop* que permite a integração de diferentes espaços de armazenamento em *cloud* num único espaço, de forma transparente. O utilizador passa a dispor de um espaço de armazenamento integrado e a aplicação encarrega-se de gerir da forma mais conveniente esse espaço, sendo da responsabilidade desta escolher em que *cloud* (ou *clouds*) alojar os ficheiros. Permite, inclusive, alojar ficheiros que excedam em tamanho o tamanho livre numa única *cloud*, fragmentando o ficheiro e distribuindo os fragmentos pelas várias *clouds*. O utilizador, localmente, continua a ver o ficheiro como um todo, quer na máquina onde o ficheiro foi criado, quer em outras sincronizadas com as mesmas *clouds*.

A solução integrada foi implementada usando as *clouds* Dropbox e Google Drive. A sua escolha assentou no facto de serem as mais populares entre os utilizadores domésticos. Foi originalmente planeada a implementação da solução usando 3 serviços de armazenamento em *cloud*. No entanto, devido a constrangimentos temporais não houve oportunidade para incluir o terceiro serviço. Deste modo, foram apenas usados dois, pois era o mínimo necessário para que houvesse uma solução integrada. Assim, apesar da implementação final ficar aquém das expetativas, a solução obtida é uma prova de conceito válida para aquilo que eram os objetivos da dissertação.

O último ponto dos objetivos pressupunha a construção de uma interface gráfica que permitisse a configuração da aplicação, nomeadamente para permitir a sincronização seletiva e o arranque e a paragem da sincronização, entre outras funcionalidades. A interface não chegou a ser desenvolvida, embora as funcionalidades estejam total ou parcialmente concluídas.

Relativamente à política de fragmentação, adotou-se uma política cujos fragmentos são de tamanho fixo. Pensou-se em usar fragmentos de tamanho variável, principalmente porque os envios para as *clouds*, como referido no capítulo 2, também são feitos em *chunks*. Desta forma, na fragmentação, os ficheiros poderiam ser divididos em fragmentos de tamanho diferente consoante a *cloud* destino. Devido a algumas limitações nos métodos disponibilizados pelas APIs oficiais (mencionados em 2.1.2), não se pode fazer essa fragmentação em memória, obrigando à criação dos fragmentos em disco. No entanto, no envio de dados, a divisão em *chunks* é obrigatoria por parte das APIs oficiais, de uma forma mais

eficiente que a criação dos fragmentos em disco.

A versão atual da aplicação precisa de ser submetida a mais testes de robustez, com o objetivo de identificar e eliminar possíveis pontos de falha. Em particular, há cenários de utilização mais propensos à ocorrência de falhas. Situações de circularidade, falhas de conexão a uma das *clouds* e manipulação simultânea do mesmo ficheiro em máquinas diferentes são cenários onde há mais probabilidade do sistema poder falhar e que, por isso, precisam de testes de robustez adequados.

No caso da Dropbox, foi referido no capítulo 4 que a API utilizada é uma versão beta. À data do desenvolvimento, a versão 2 da API estava também em desenvolvimento. Ainda assim, fez sentido adotar-se a versão 2, embora em versão beta, pois as alterações para uma versão final seriam previsivelmente menores do que partir de uma versão anterior onde os métodos são muito diferentes. Consequentemente, assim que seja possível, é necessário mudar a API para a versão final e fazer as alterações necessárias para que a aplicação fique mais estável.

A base de dados de ficheiros fragmentados deve estar replicada em todos os serviços *cloud* para que cada *cloud* tenha a informação mais atualizada possível. Na versão atual da aplicação, apenas a Dropbox gere essa base de dados. Isto implica que para uma boa deteção e reconstrução de ficheiros fragmentados a aplicação se deve conectar em primeiro lugar à Dropbox. O mecanismo de deteção e reconstrução desenrola-se em dois passos: primeiro deve ser carregado a base de dados e só depois devem ser carregados os fragmentos. Ou seja, o bom funcionamento da versão atual da aplicação depende do tempo de reação da Dropbox a notificar os outras máquinas da nova versão da base de dados. No futuro, deve ser implementado um método que permita garantir por completo a boa deteção de ficheiros fragmentados.

Outro fator a ter em conta, ainda relativamente à base de dados de ficheiros fragmentados, é que não é garantido o isolamento quando duas máquinas tentam modificar essa mesma base de dados simultaneamente. Ou seja, duas máquinas podem, ao mesmo tempo, fazer *upload* de dois ficheiros diferentes que se vão fragmentar. Nessa situação particular, podem tentar modificar a mesma base de dados ao mesmo tempo. Como apenas confiamos nos mecanismos de cada *cloud*, estes não asseguram consistência neste tipo de situações. Como tal é necessário assegurar que a base de dados fica consistente a todos os momentos.

No que diz respeito à política de distribuição de ficheiros, foi utilizado o balanceamento equitativo do espaço livre das *clouds*. Porém, podem-se considerar outras políticas, como por exemplo, o balanceamento equitativo do espaço ocupado ou um balanceamento proporcional desse espaço. A aplicação desenvolvida pode ser estendida incorporando estas novas políticas de distribuição ou outras e permitir que o utilizador escolha aquela que prefere ou mais lhe convém.

Bibliografia

[1] Enrico Bocchi, Idilio Drago, and Marco Mellia. Personal cloud storage: Usage, performance and impact of terminals. In *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, pages 106–111. IEEE, 2015.

[2] Moritz Borgmann, Tobias Hahn, Michael Herfert, Thomas Kunz, Marcel Richter, Ursula Viebeg, and Sven Vowe. On the security of cloud storage services. 2012.

[3] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 892–903, New York, NY, USA, 2014. ACM.

[4] Matt Dee. Inside LAN sync — dropbox tech blog. `https://blogs.dropbox.com/tech/2015/10/inside-lan-sync/`, October 2015. (Acedido a 6 de novembro de 2015).

[5] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 205–212, New York, NY, USA, 2013. ACM.

[6] Idilio Drago, Marco Mellia, Maurizio M. Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: Understanding personal cloud storage services. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 481–494, New York, NY, USA, 2012. ACM.

[7] Dropbox, Inc. Developers - dropbox: OAuth guide. `https://www.dropbox.com/developers/reference/oauth-guide`, 2016. (Acedido a 26 de janeiro de 2016).

[8] Dhiru Kholia and Przemysław Węgrzyn. Looking inside the (drop) box. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2013. USENIX.

[9] Google, Inc. Authorizing your app with google drive. `https://developers.google.com/drive/v2/web/about-auth`, January 2016. (Acedido a 10 de fevereiro de 2016).

[10] Google, Inc. Google trends - interesse em pesquisa web do google: Dropbox, google drive, onedrive, box, owncloud - a nível mundial, 2004 - presente. https://goo.gl/eqkzvt, June 2016. (Acedido a 4 de junho de 2016).

[11] Google, Inc. Using OAuth 2.0 to access google APIs. https://developers.google.com/identity/protocols/OAuth2, January 2016. (Acedido a 13 de fevereiro de 2016).

[12] Eran Hammer. Introducing OAuth 2.0. https://hueniverse.com/2010/05/15/introducing-oauth-2-0/, May 2010. (Acedido a 2 de abril de 2016.).

[13] Jakob Jenkov. Java NIO vs. IO. http://tutorials.jenkov.com/java-nio/nio-vs-io.html, June 2014. (Acedido a 20 de outubro de 2015).

[14] Barry Leiba. Oauth web authorization protocol. *IEEE Internet Computing*, 16(1):74–77, 2012.

[15] Ben Martini and Kim-Kwang Raymond Choo. Cloud storage forensics: Owncloud as a case study. *Digit. Investig.*, 10(4):287–299, December 2013.

[16] Jakub T Mościcki and Massimo Lamanna. Prototyping a file sharing and synchronization service with owncloud. *Journal of Physics: Conference Series*, 513(4):042034, 2014.

[17] Roberto Eli Sanmartim. Cloud computing-estudo de caso: ferramentas de armazenamento. 2014.

[18] J. Sendor, Y. Lehmann, G. Serme, and A. Santana de Oliveira. Platform-level support for authorization in cloud services with oauth 2. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 458–465, March 2014.

[19] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: An empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 378–390, New York, NY, USA, 2012. ACM.

[20] Mohamed Taman. JDK7: Part 1- the power of java 7 NIO.2 (JSR 203) (important concepts). http://tamanmohamed.blogspot.com/2012/03/jdk7-part-1-power-of-java-7-nio2-jsr.html, March 2014. (Acedido a 23 de outubro de 2015).

[21] Jack Wallen. Five tools to help sync more than one cloud storage service. http://www.techrepublic.com/blog/five-apps/five-tools-to-help-sync-more-than-one-cloud-storage-service/, May 2013. (Acedido a 3 de março de 2016).